מכון טכנולוגי חולון
Holon Institute of Technology

Distributed Software Systems – Principles and Practical Applications

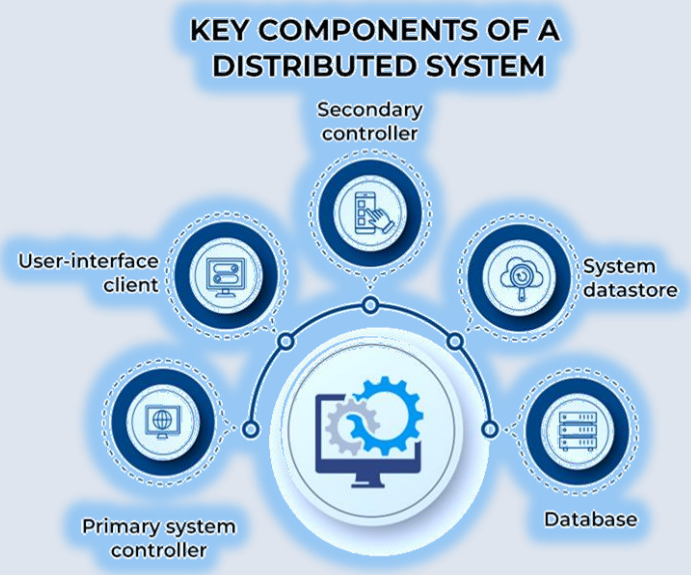
Self-stabilizing Algorithms in Distributed Systems

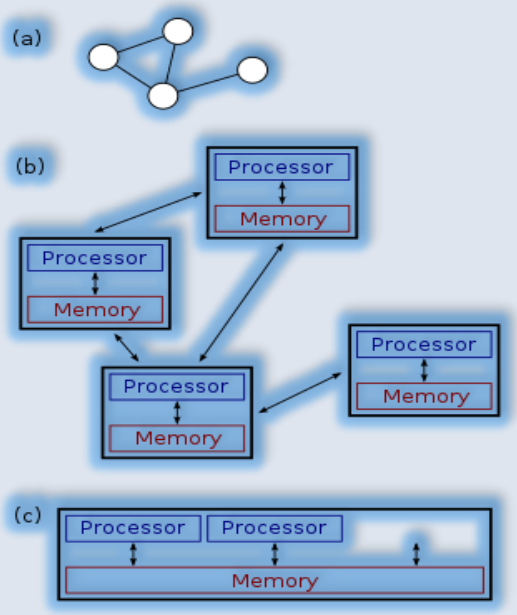
Final Semestrial (Semester B) Project

Course Number: 667024
Student Name: Or Attias
Student ID: 207953308
Course Lecturer: Pr. Iaakov Exman

June 2023

Computer Science Department
Faculty of Science, Holon Institute of Technology





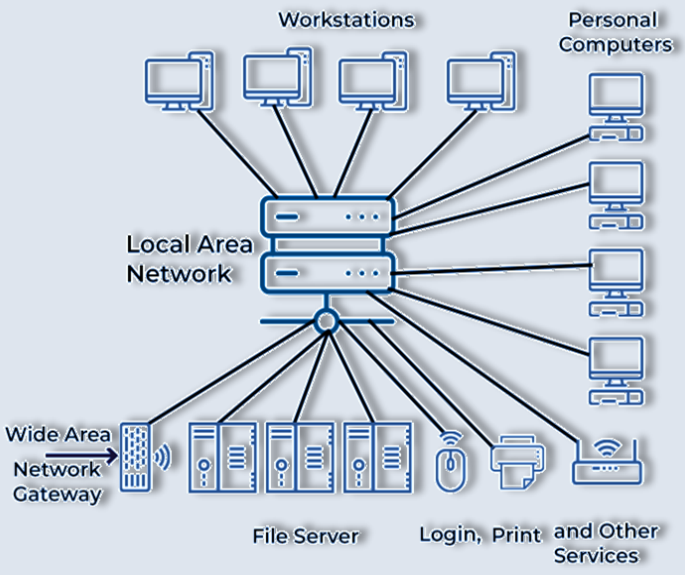


Table of Contents

The Goal of the project

The goal of this final project is to analyze self-stabilization in distributed software systems, with a specific focus on the census problem.

Distributed systems are prevalent in today's technological landscape, powering a wide range of applications and services.

However, ensuring the correctness and stability of such systems can be challenging due to the presence of faults and unpredictable behavior. Self-stabilization algorithms offer a promising approach to address these challenges by enabling systems to recover and maintain correct behavior in the presence of transient faults and dynamic changes.

The project will focus on understanding the principles and practical applications of self-stabilization algorithms, with a specific emphasis on the census problem. The census problem involves determining the global state or configuration of a distributed system, even when individual components may be experiencing faults or undergoing updates. By studying self-stabilization algorithms for census, I aim to develop a deeper understanding of their theoretical foundations and practical implications.

Throughout the project, I will explore various self-stabilization algorithms proposed in the literature and evaluate their effectiveness in achieving correct and stable census results. I will investigate different strategies for fault detection, fault tolerance, and system recovery, as well as their impact on system performance and scalability.

Additionally, I will analyze the resilience and adaptability of these algorithms in the face of changing network conditions and component failures.

By the end of the project, I aim to provide a comprehensive overview of self-stabilization algorithms for census in distributed software systems, highlighting their strengths, limitations, and potential areas for further research.

The insights gained from this project will contribute to the broader field of distributed systems and provide valuable knowledge for designing robust and fault-tolerant software architectures. Through this exploration of self-stabilization algorithms for census, I hope to deepen my understanding of the principles behind these algorithms and their practical applications in real-world distributed software systems.

1) Fault Taxonomy in Distributed Systems

Faults Classification

1st Criteria - localization in time

1. **transient faults:** faults that are arbitrary in nature can strike the system, but there is point in the execution beyond which these faults no longer occur;
2. **permanent faults:** faults that are arbitrary in nature can strike the system, but there isa point in the execution beyond which these faults always occur;
3. **intermittent faults:** faults that are arbitrary in nature can strike the system, at any moment in the execution.

2nd criteria - nature of the faults

An element of the distributed system can be represented by an automaton, whose states represent the possible values of the element's variables, and whose transitions represent the code run by the element. We can then distinguish the following faults depending on whether they involve the state or the code of the element:

1. **state related faults:** changes in an element's variables may be caused by disturbances.in the environment (electromagnetic waves, for example), attacks or simply faults on the part of the equipment used. For example, it is possible for some variables to have values that they are not supposed to have if the system is running normally.

2. **code-related faults:** an arbitrary change in an element's code is most often the result of an attack (the replacement, for example, of an element by a malicious opponent), but certain, less serious types correspond to bugs or a difficulty in handling the load. There are several different sub-categories of code-related faults:
   - **crashes:** at a given moment during the execution, an element stops its execution permanently and no longer performs any action.
   - **omissions:** at different moments during the execution, an element may omit to communicate with the other elements of the system, either in transmission, or in reception.
   - **duplications:** at different moments during the execution, an element may perform an action several times, even though its code states that this execution must be performed once.
   - **de-sequencing:** at different moments during the execution, an element may perform the right actions, but in the wrong order.
   - **Byzantine faults:** these simply correspond to an arbitrary type of fault, and are therefore, the faults that cause the most harm.

2) Self-Stabilization Algorithms

a) Definition

Starting from an arbitrary initial configuration, any ex- ecution of a self-stabilizing algorithm contains a subsequent configuration from which every execution satisfies the specification.

b) Self-Stabilization Algorithms Classification

In terms of distributed software systems, self-stabilization algorithms can be categorized based on the following criteria:

1. **Network Model:** Self-stabilization algorithms can be classified based on the network model they are designed for. For example, some algorithms are designed for a complete network topology, while others are designed for a partial network topology.
2. **Communication Model:** Self-stabilization algorithms can also be classified based on the communication model used in the system. For example, some algorithms assume a synchronous communication model, while others assume an asynchronous communication model.
3. **Fault Model:** Self-stabilization algorithms can be classified based on the fault model they are designed to handle. For example, some algorithms are designed to handle only transient faults, while others can handle Byzantine faults.
4. **Algorithmic Technique:** Self-stabilization algorithms can also be classified based on the algorithmic technique they use. For example, some algorithms are based on token passing, while others are based on message-passing or graph-based approaches.

Overall, the categorization of self-stabilization algorithms for distributed software systems helps researchers and practitioners understand the different approaches and techniques used in self-stabilization and select the most appropriate algorithm for their specific system and fault scenario.

c) Fault-tolerant Algorithm & Categories

When faults occur on one or several of the elements that comprise a distributed system, it is essential to be able to deal with them.

If a system tolerates no fault whatsoever, the failure of a single-one of its elements can compromise the execution of the entire system: this is the case for a system in which an entity has a central role (such as the DNS). To preserve the system's useful lifespan, several ad hoc methods have been developed, which are usually specific to a particular type of fault that is likely to occur in the system in question. However, these solutions can be categorized depending on whether the effect is visible or not to an observer (a user, for example). A masking solution hides the occurrence of faults to the observer, whereas a non-masking solution does not present this characteristic: the effect of faults is visible over a certain period, then the system resumes behaving properly.

A masking approach may seem preferable at first, since it applies to a greater number of applications. Using a non-masking approach to regulate air traffic would make collisions possible following the occurrence of faults. However, a masking solution is usually more costly (in resources and in time) than a non-blocking solution and can only tolerate faults so long as they have been anticipated. For problems such as routing, where being unable to transport information for a few moments will not have catastrophic consequences, a non- masking approach is perfectly well-suited.

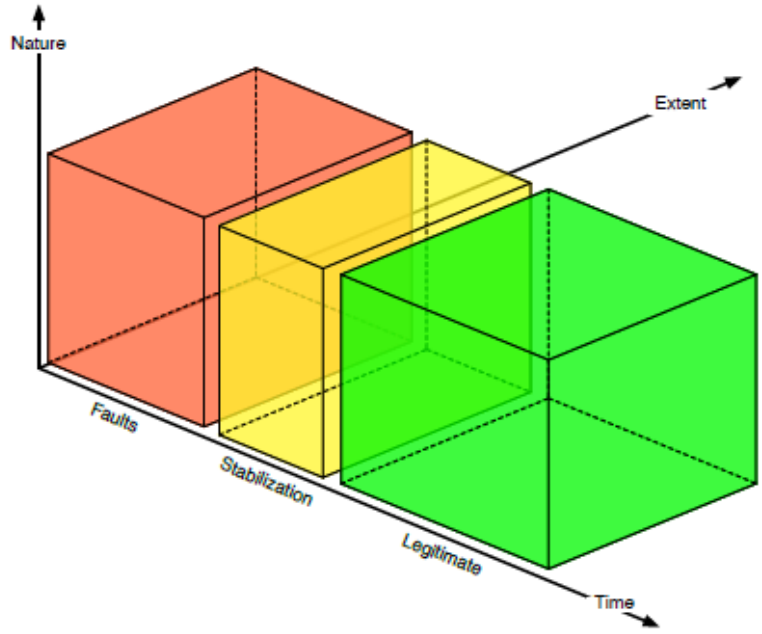
The masking approach is a method of hiding the occurrence of faults from the observer, such as a user or operator. In this approach, when a fault occurs, the system continues to operate normally without any visible signs of the fault. Instead, the fault is detected and repaired behind the scenes without any interruption to the system's functioning. This approach is useful in some applications where it is critical to ensure the system appears to operate without any failures, even if faults occur. However, masking solutions are often more complex and resource-intensive than non-masking solutions and can only tolerate anticipated faults.

Two major categories for fault-tolerant algorithms can be distinguished:

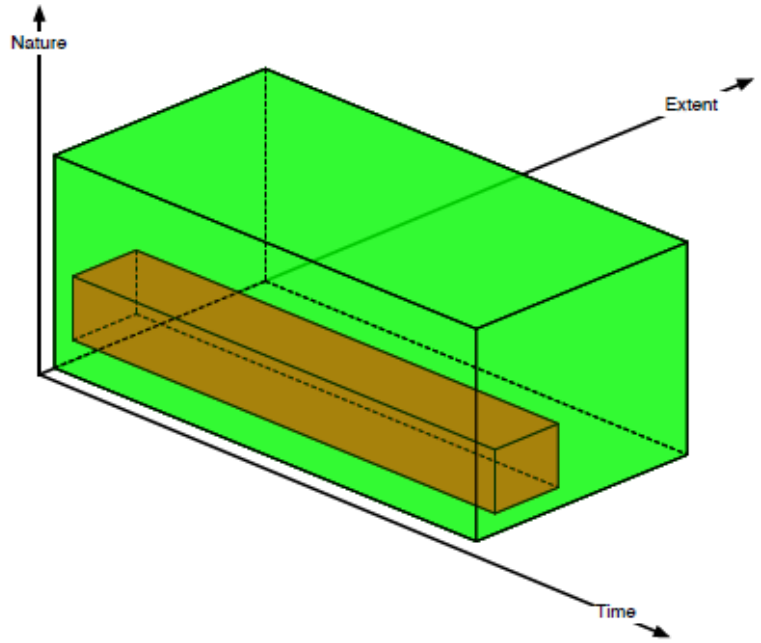
1. ***robust algorithms:*** these use redundancy on several levels of information, of communications, or of the system's nodes, in order to overlap to the extent that the rest of the code can safely be executed. They usually rely on the hypothesis that a limited number of faults will strike the system, so as to preserve at least a majority of correct elements (sometimes more if the faults are more severe). Typically, these are masking algorithms.
2. ***self-stabilizing algorithms:*** these rely on the hypothesis that the faults are transient (in other words, limited in time), but do not set constraints regarding the extent of the faults (which may involve all the system's elements). An algorithm is self-stabilizing if it manages, in a finite time, to present an appropriate behavior independently from the initial state of its elements, meaning that the variables of the elements may exist in a state that is arbitrary (and impossible to achieve by running the application normally). When a fault occurs in a system that uses a self-stabilizing algorithm, the algorithm will work to restore the system to a stable state. However, during the process of restoring the system, the system may behave erratically or unpredictably before it stabilizes to an appropriate behavior. This means that the effects of the fault may not be hidden from observers or users of the

system, unlike in masking algorithms where the effects of faults are hidden from users. Hence, self-stabilizing algorithms are typically non-masking.

Fault-tolerant Algorithms Visualizations:



(a) Self-stabilizing algorithms



(b) Robust algorithms

Figure 1: Self-stabilization *vs.* Robustness

I will be focusing on Self-Stabilization Algorithm in distributed software system while exploring the Census Self-Stabilization algorithm.

3) System Hypotheses

In the context of self-stabilization, the hypotheses made for the system generally do not include, as with robust algorithms, conditions on the completeness or the globality of the communications. Many algorithms run on systems with nodes that only communicate locally.

However, several hypotheses may be crucial for the algorithm to run properly, and involve the hypotheses made regarding the scheduling of the system:

1. **atomicity of the communications:** use communication primitives with a high level of atomicity. At least three historic models are found in the literature.
   - (a) the state model (or shared memory model
   - (b) the shared register models: in one atomic step, a node can read the state of one.
     of its neighboring nodes, or update its own state, but not both simultaneously.
   - (c) the message passing model: for which in one atomic step, a node sends a message to one of the neighboring nodes or receives a messages from one of the neighboring nodes, but not both simultaneously.

The literature shows examples of how to transform one communication model to another. For instance, some researchers have demonstrated how to transform the local diffusion model with collisions into a shared memory model, while others have shown how to transform a specific model into a shared memory model. However, such transformations come with two issues:

- (a) Using the transformation method between models requires additional resources such as time, memory, and energy, which could be avoided by using a direct solution in the model that is closest to the actual system being considered.
- (b) The transformation method can only be applied in systems with bidirectional communications since acknowledgments must be regularly sent to ensure that the highest-level model is properly simulated.

2. **spatial scheduling:**
   - (a) central scheduling: at a given moment, only one of the system's nodes can run its code.
   - (b) global (or synchronous) scheduling: at a given moment, all the system's nodes run their codes.
   - (c) distributed scheduling: at a given moment, an arbitrary subset of the system's nodes run its code. This type of spatial scheduling is the most realistic.

3. **temporal scheduling:**
   - (a) arbitrary (aka unfair, adversarial) scheduling:
   - (b) fair scheduling:
   - (c) bounded scheduling:

4) Taxonomy of System Hypotheses in Self-stabilization

Explanation:

1. **Temporal Scheduling:** This hypothesis assumes that there is some kind of time delay between the moment a process fails and the moment it can resume its normal operation. In other words, this hypothesis assumes that processes need some time to recover after a failure.

2. **Atomicity Hypothesis:** This hypothesis assumes that each operation performed by a process must be completed in its entirety before any other operation can be executed. This ensures that the system is always in a consistent state and prevents inconsistencies caused by partial operations.

3. **Spatial Scheduling:** This hypothesis assumes that processes are only allowed to access specific resources or parts of the system at certain times. This helps prevent conflicts and ensures that processes do not interfere with each other.

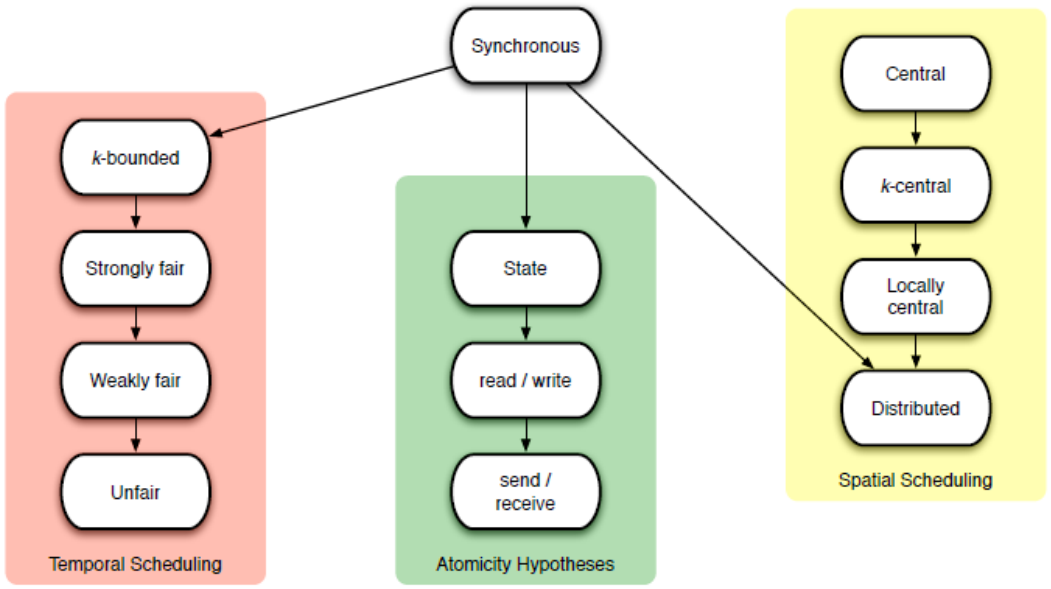


Figure 2: Taxonomy of system hypotheses in Self-stabilization

5) Designing self-stabilizing Algorithms

Depending on the problem that we wish to solve, the minimum time required for going back to a correct configuration varies significantly.

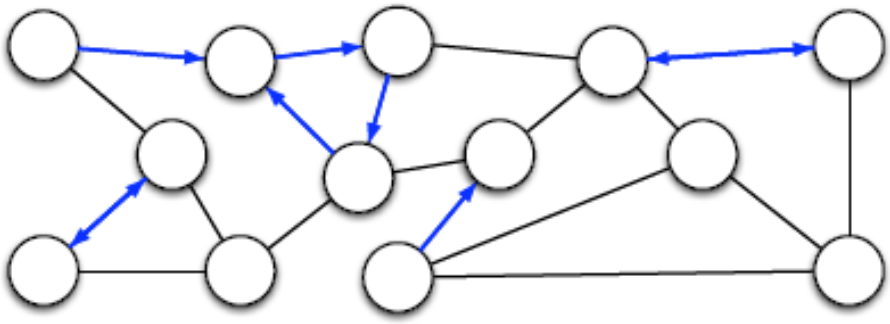
Problems are generally divided into two categories:

1. **static problems:** we wish to perform a task that consists of calculating a function that depends on the system in which it is assessed. For example, it can consist of coloring the nodes of a network to never have two adjacent nodes with the same color; another example is the calculation of the shortest paths to a destination.
2. **dynamic problems:** we wish to perform a task that performs a service for upper layer algorithms. The model transformation algorithms such as token passing fall into this category.

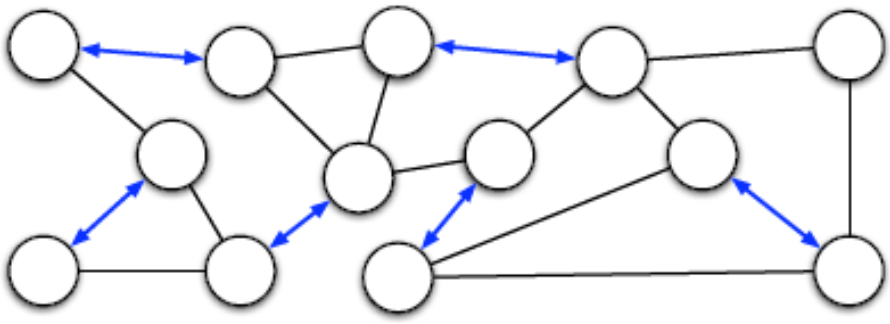
Maximal Matching

Algorithm for computing a maximal matching. The algorithm is self-stabilizing and does not make any assumptions on the network topology.

Algorithm 1 MM: a self-stabilizing maximal matching algorithm



(a) A possible arbitrary initial configuration



(b) A legitimate maximal matching configuration

Figure 3: Possible configurations of Algorithm 1

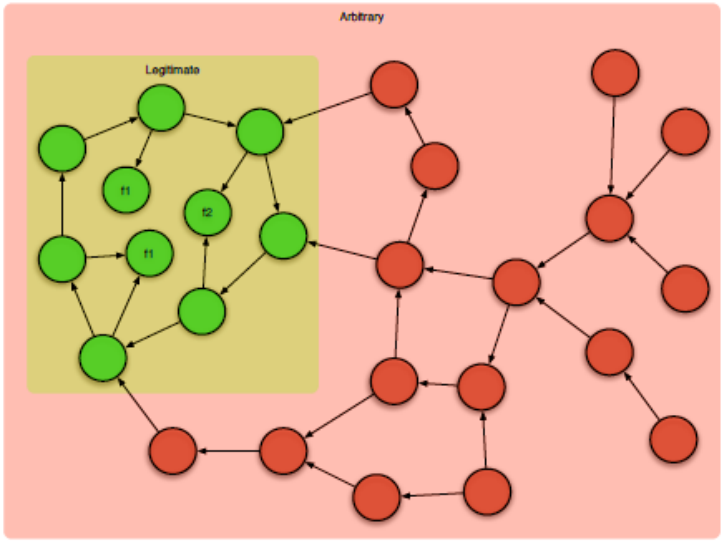
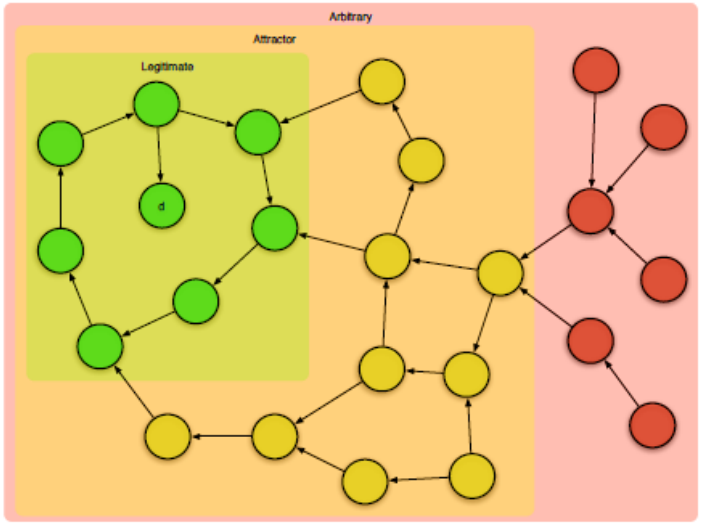
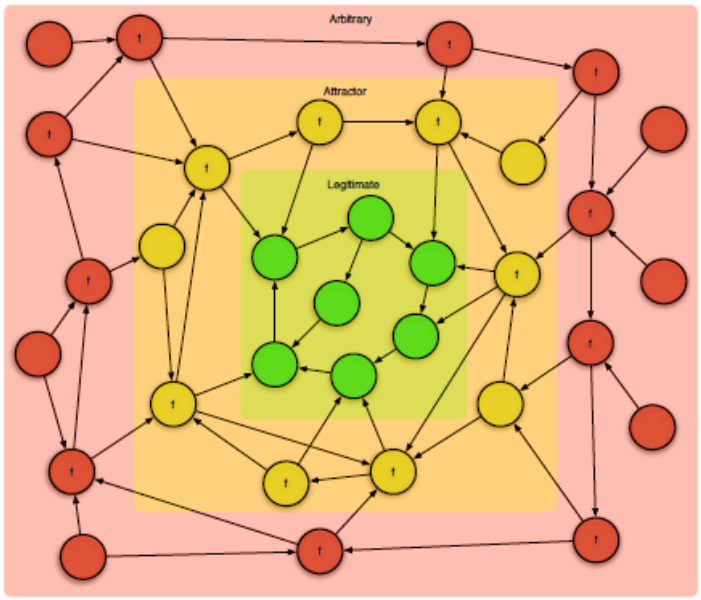


Figure 4: Proving Self-stabilization with unfair scheduling



(a) without fairness condition



(b) with fairness condition

Figure 6: Proving Self-stabilization with attractors

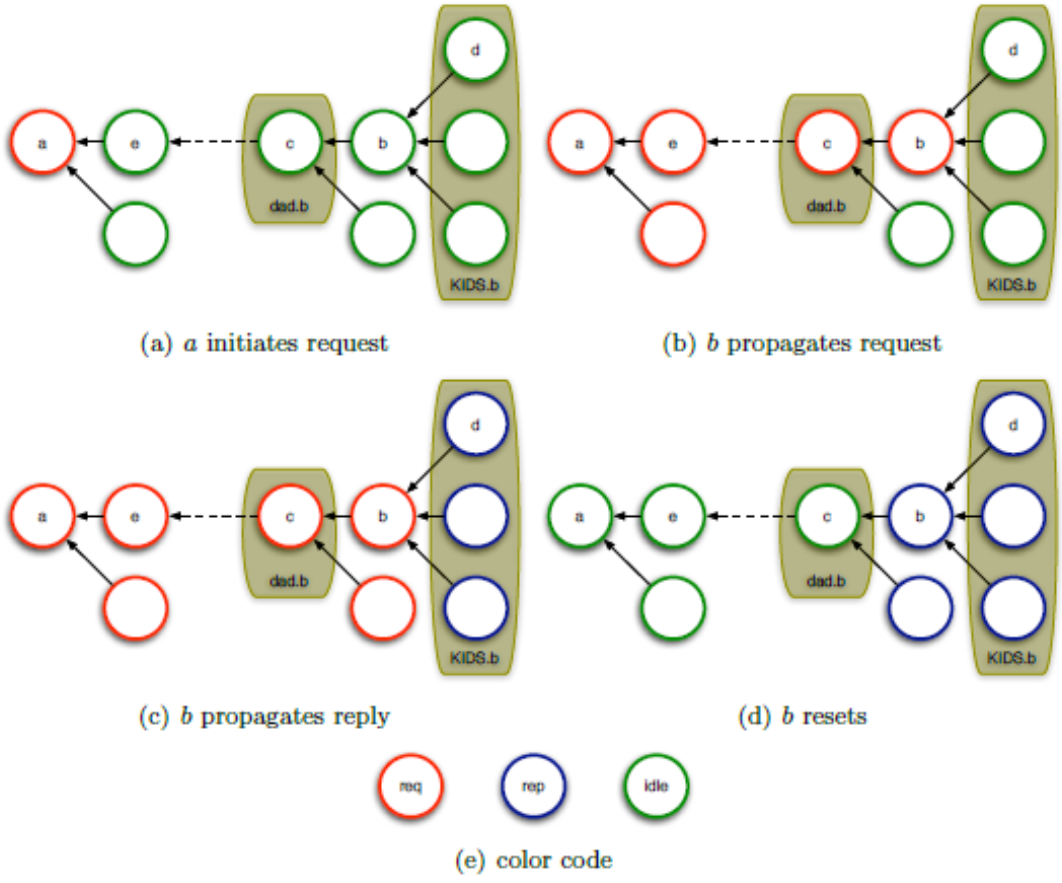
Algorithm 2 KDP: a self-stabilizing k-hops diners' algorithm

Algorithm 2 is a self-stabilizing solution to the k-hop diners' problem.

A program that solves the generalized diners satisfies the following two properties for each process p:

safety — if the action that executes the CS is enabled in p, it is disabled in all processes of $M.p$;

liveness — if p wishes to execute the CS, it is eventually allowed to do so.



(a) a initiates request

(b) b propagates request

(c) b propagates reply

(d) b resets

(e) color code

Figure 5: Phases of Algorithm 2 operation

Algorithm 2 is a self-stabilizing solution to the k-hop diners problem.

6) Census self-stabilizing Algorithm

a) Background:

In their article " Tolerating Transient and Intermittent Failures", Sylvie Delaët, Sébastien Tixeuil presented a self-stabilizing census algorithm.

for strongly connected directed networks. The census task requires each processor to have a list of all processor identifiers and their relative distance.

The algorithm presented as a distributed census algorithm for unidirectional rings subject to transient faults.

b) Definition:

This algorithm is written using the efficient cut-through routing scheme, where the messages must be forwarded to a neighbor before they are completely received.

The distributed census problem can be informally described as follows:

The nodes cooperate to reach a global configuration where every node can determine, within finite time, which nodes are present in the network and which nodes are not present in the network. The fault-tolerance is achieved using Dijkstra's paradigm of self-stabilization.

A self-stabilizing algorithm, regardless of the initial system configuration, converges in finite time to a set of legitimate configurations. Some fixed output algorithms work by accumulating information.

In a census algorithm, each process gradually learns new process identifiers until it knows them all.

On the other hand, the algorithm must make sure that if a process knows an identifier that does not exist, then this identifier is eventually removed.

c) Initial Configuration:

In this context, an initial configuration should be a minimum-information configuration.

In a census algorithm, an initial configuration is one where each process knows no other process, except itself.

Similarly, in an initial configuration, no process has chosen a new name in a renaming algorithm or a color in a coloring algorithm, no process is part of the topology, or particular set which being built in a graph algorithm.

The fact that initial configurations put a fixed output algorithm A in the beginning of an execution where it will not acquire false information means that an algorithm using the output of A will not make wrong moves. It thus brings additional safety to the system.

Resetting a system to an initial configuration may thus be appropriate in situations where the system as a whole should behave conservatively, i.e. when mistakes have a high cost

Census Algorithm – Fundamentals:

1. Knowledge Objects & Network Nodes

- **Each node *i*** has a **unique identifier** and is aware of its **input degree $\delta^-.i$** (the number of its incident arcs), which is also placed in non-corruptible memory.
  - **A node *i* arbitrarily numbers** its incident arcs using the first $\delta^-.i$ natural numbers. When receiving a message, the node *i* knows the number of the corresponding incoming link (that varies from 1 to $\delta^-.i$).

- Each node maintains a **local memory.**
  - The local memory of ***i*** is represented by a list denoted by $(i_1; i_2; \ldots; i_k)$.
  - **Each i_α** is a non-empty list of pairs *(identifier, colors)*, where ***identifier*** is a node identifier, and where ***colors*** is an array of Booleans of size $\delta^-.i$.

- Each Boolean in the *colors* array is either ***true*** (**denoted by •**) or ***false*** (**denoted by ◦**). We assume that natural operations on Boolean arrays, such as unary *not* (denoted by ¬), binary *and* (denoted by ∧) and binary *or* (denoted by ∨) are available.
  - **Coherency** is implemented within this array: with every message received, each Boolean indicates **whether** this information was received through incoming links represented.

The goal of the Census algorithm is to **guarantee that the local memory of each node contains the list of lists of identifiers** (whatever the *colors* value in each pair *(identifier, colors)*) that are predecessors of *i* in the communication graph. Each predecessor of *i* is present only once in the list.

2. Message Example

Information from z is carried through all three incoming links (or "arcs") to node i, which indicates that node z has outgoing links to all three nodes that are direct ancestors of i (namely, j, q, and t).

Each node sends and receives messages.

The contents of a message are represented by a list denoted by $(i_1; i_2; \ldots; i_k)$.

Each i_α is a non-empty list of *identifiers*.

For example,

$((i)(j; q; t)(z))$

is a possible content of a message.

It means that node *i* sent / emitted the message (since it appears first in the message), that *i* believes that *j*, *q*, and *t* are the direct ancestors of *i*, and that *z* is an ancestor at distance 2 of *i*.

3. Local Memory of Node in the Network

I will show an example of local memory of node i, as explained above.

$$((j, [\bullet \circ \circ]; q, [\circ \bullet \circ]; t, [\circ \circ \bullet])(z, [\bullet \bullet \bullet]))$$

Message: ((i)(j; q; t)(z))

This represents the local memory of node **i**.

Recall that **i** has **input degree** $\delta^-.i$ equal to **3**, meaning that it has **3** incoming communication links (also known as "arcs").

The first list in **i**'s local memory is **[(j, [• ∘ ∘]); (q, [∘ • ∘]); (t, [∘ ∘ •])]**.

Each item in the list is a pair, where the first element is a node identifier (**j**, **q**, or **t**) and the second element is a **3**-bit Boolean array.

For example, the first item in the list is **(j, [• ∘ ∘])**.

This indicates that node **j** is a direct ancestor of **i** in the communication graph, and the first incoming link (indexed by **1**) carries information from **j** to **i**.

The Boolean array **[• ∘ ∘]** indicates that the information received from **j** over the first incoming link is only relevant to the first position in the array (since the first position is • and the other two are ∘).

Similarly, the second item in the list is **(q, [∘ • ∘])**, indicating that node **q** is also a direct ancestor of **i** in the communication graph, and the second incoming link (indexed by **2**) carries information from **q** to **i**.

The Boolean array **[∘ • ∘]** indicates that the information received from **q** over the second incoming link is only relevant to the second position in the array.

Finally, the third item in the list is **(t, [∘ ∘ •])**, indicating that node **t** is a direct ancestor of **i** in the communication graph, and the third incoming link (indexed by **3**) carries information from **t** to **i**.

The Boolean array **[∘ ∘ •]** indicates that the information received from **t** over the third incoming link is only relevant to the third position in the array.

The second list in **i**'s local memory is **[(z, [• • •])]**. This indicates that node **z** is **2** hops away from **i** in the communication graph, and information from **z** is carried through all three incoming links (since all positions in the Boolean array **[• • •]** are •).

This means that node i has received a message from node z through all three incoming links. The Boolean array [• • •] in the pair (z, [• • •]) means that all the bits are true, indicating that node **z** is reachable from node i through all three incoming links. This

also means that node z is 2 hops away from node i in the communication graph since the message was received through all three incoming links. The Boolean array [• • •] indicates that the information received from z over all three incoming links is relevant to all three positions in the array.

4. Census Algorithm Functions

To improve the readability of our algorithm, we will introduce helper functions that work with lists, integers, and pairs of (identifier, colors). These functions are specified using the following notations:

Knowledge Objects:

1) "l" represents a list of identifiers.
    a. **Example:** l = [1, 2, 3, 4, 5]
2) "p" represents an integer.
    a. **Example:** p = 3
3) "lc" represents a list of pairs of (identifier, colors).
    a. **Example:** lc = [(1, True), (2, False), (3, True), (4, False), (5, True)]
4) "Ll" represents a list of lists of identifiers.
    a. **Example:** Ll = [[1, 2], [3, 4, 5], [6, 7, 8, 9]]
5) "Llc" represents a list of lists of pairs of (identifier, colors).
    a. **Example:** Llc = [[(1, True), (2, False)], [(3, True), (4, False), (5, True)], [(6, False), (7, True), (8, False), (9, True)]]

Functions:

Name	Parameters	Returns	Main purpose	Principle in use
colors(p)	p: integer	Array of booleans	**Returns the array of booleans that correspond to the pth incoming link**	Trust Most Recent Information
clean(lc, p)	lc, p: integer	List of pairs (identifier, color)	**Returns the list of pairs that have colors that are not equal to [◦ · · · ◦] on the pth incoming link**	Check Local Coherence
emit(i, Llc)	i: identifier, Llc	None	**Sends the message resulting from (i)+identifiers(Llc) on every outgoing link of i**	N/A
identifiers(Llc)	Llc	List of lists of identifiers	**Returns the list of lists of identifiers where each pair (identifier,colors) in Llc becomes identifier in Ll**	N/A
merge(lc, l, p)	Lc , p	List of pairs (identifier, colors)	**Returns the list of pairs that have identifiers in lc and l and merges the colors based on path incoming link**	Trust Most Recent Information
new(lc, l)	lc:	List of pairs (identifier, colors)	**Returns the list of pairs that have identifiers in lc but not in l**	Check Local Coherence
problem(Llc)	Llc	Boolean	**Returns true if there exist two integers p and q such that Llc[p][q] is of the form (identifier, colors) and all Booleans in colors are false (◦)**	Check Message Coherence

5. Census Algorithm – Step Rules:

1. **Check Message Coherence:**

Since all nodes have the same behavior, when a node receives a message that does not start with a singleton list, the message is trivially considered as erroneous and is ignored.

For example, messages of the form ((j; q; t)(k)(m; y)(p; z)) are ignored.

2. **Check Local Coherence:**

To ensure local coherence, each node regularly checks for any trivial inconsistencies in its local memory. Specifically, it checks for the existence of at least one pair of (identifier, colors) such that colors = [◦ · · · ◦], which indicates that some information in the local memory was not obtained from any of the input channels. If a problem is detected upon time-out, **then the local memory is reinitialized.** If problem is detected upon a message receipt, the local memory is **completely replaced by the information contained in the message.**

Overall, the Check Local Coherence step helps maintain the integrity of the census algorithm by ensuring that all nodes have consistent information and can make accurate decisions about the network.

For example,

Scenario 1: If a problem with local coherence is detected upon time-out

- Suppose we have a network of 5 nodes, and each node has 3 input channels to receive information from its neighbors.
- After some time, node 2 detects a problem with its local coherence because one of its colors array is [◦ · · · ◦], meaning that it is missing some information from its input channels.
- Since this problem was detected upon time-out, node 2 will reinitialize its local memory by resetting its colors array to [• • •]. ([◦ ◦◦] will not change the array)

Scenario 2: If a problem with local coherence is detected upon receiving a message

- Suppose we have a network of 4 nodes, and each node has 2 input channels to receive information from its neighbors.
- Node 1 receives a message from node 2, but detects a problem with its local coherence because one of its colors array is [◦ ◦ · ·], meaning that it is missing some information from its input channels.
- Since this problem was detected upon message receipt, node 1 will replace its entire local memory with the information contained in the message from node 2, including the colors array.

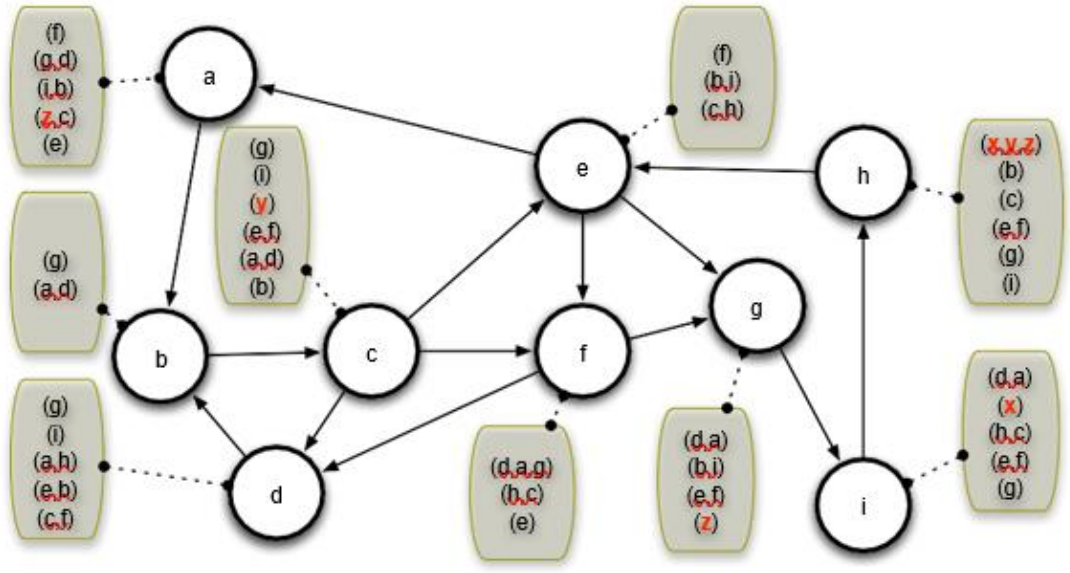
3. **Trust Most Recent Information:**

When a node receives a message through an incoming channel, this message is expected to contain more recent and thus more reliable information. The node removes all previous information obtained through this channel from its local memory. Then it integrates new information and only keeps old information (from its other incoming channels) that does not clash with new information.

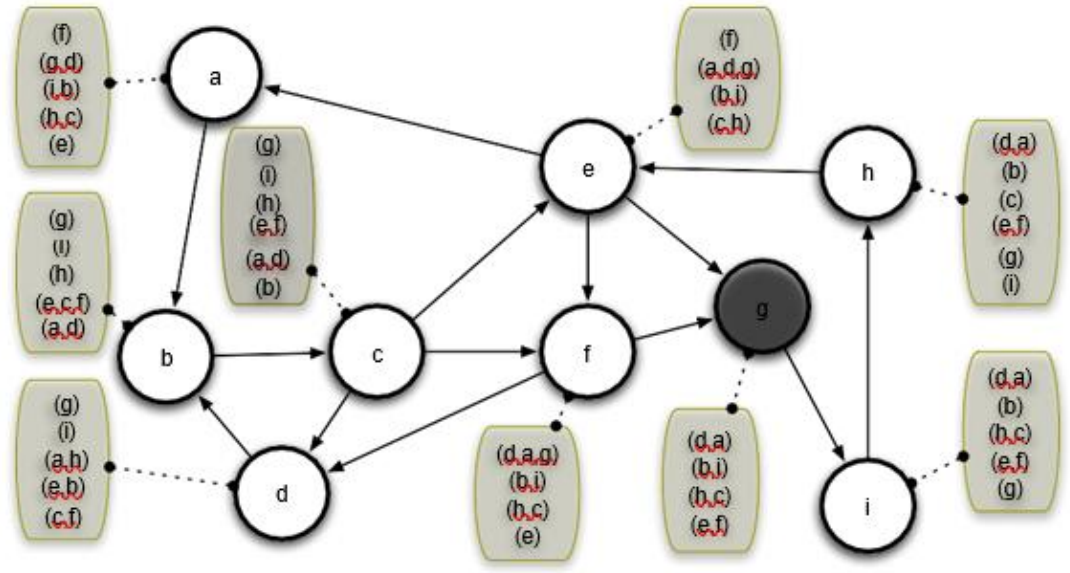
7) Census Network Initial Configurations

Possible corrupted initial configuration: in subfigure (a), where some information is missing and some information is incorrect, e.g. nodes x, y, and z are non-existent

Legitimate configuration (in subfigure (b), where the link contents are not displayed.



(a) A possible corrupted initial configuration



(b) A legitimate configuration (messages not displayed)

Figure 8: Two possible configurations for Algorithm 3

6. Knowledge Components Extensions of Census Algorithm

Our algorithm can be seen as a knowledge collector on the network.

Knowledge Components	Extension
The local memory of a node	The current knowledge of this node about the whole network
The only certain knowledge a node may have about the network is local	its identifier, its incoming degree, the respective numbers of its incoming channels. This is the only information that is stored in non-corruptible memory.
Each node consists in updating in a coherent way its knowledge upon receipt of other process' messages	Each node communicating this knowledge to other processes after adding its constant information about the network
Coherent update consists in three kinds of actions	The first two being trivial coherence checks on messages and local memory, respectively

Proof of Correctness: Census as self-stabilization Algorithm

The intuitive reason for self-stabilization of the protocol is as follows: anytime a node sends a message, it adds up its identifier and pushes further in the list of lists included in every message potential fake identifiers. As the network is strongly connected, the message eventually goes through every possible cycle in the network, and every node on every such cycle removes its identifier from a list at a certain index (corresponding to the length of the cycle). So, after a message has visited all nodes, there exists at least one empty list in the list of lists of the message, and all fake identifiers are all included beyond this empty list. Since every outgoing message is truncated from the empty list onwards, no fake identifier remains in the system forever.

8) Census Algorithm in Distributed Software Systems

The algorithm for each node consists in updating in a coherent way its knowledge upon receipt of other process' messages and communicating this knowledge to other processes after adding its constant information about the network. More precisely, each information placed in a local memory is related to the local name of the incoming channel that transmitted this information. For example, node i would only emit messages starting with singleton list (i) and then not containing i since it is trivially an ancestor of i at distance 0. Coherent update consists in three kinds of actions: the first two being trivial coherence checks on messages and local memory, respectively.

This algorithm is written using the efficient cut-through routing scheme, where the messages must be forwarded to a neighbor before they are completely received. The distributed census problem can be informally described as follows: the nodes cooperate to reach a global configuration where every node can determine, within finite time, which nodes are and which nodes are not present in the network. The fault tolerance is achieved using Dijkstra's paradigm of self-stabilization. A self-stabilizing algorithm, regardless of the initial system configuration, converges in finite time to a set of legitimate configurations.

The census algorithm can be used for nodes in distributed systems.

In this case, each node would have a unique identifier, and the counters would be initialized with the node's identifier instead of the process's identifier.

The basic idea of the algorithm remains the same, but the terminology changes from processes to nodes.

In fact, the census algorithm is a well-known algorithm for distributed systems, and it has been used in various applications.

For example, it can be used in a distributed database system to ensure consistency and prevent conflicts among multiple nodes that are trying to modify the same data.

It can also be used in a peer-to-peer network to maintain a distributed index or to perform distributed search.

Real-life Usage of Census

some examples of real-life usage of census in distributed software systems:

1. **Distributed databases:** In a distributed database system, census can be used to collect information about the data stored across the network, including the number of records, the size of each record, and other relevant information. This information can then be used to optimize queries and ensure that the data is distributed efficiently across the network.
2. **Distributed monitoring:** In a distributed system, it is important to monitor the health and performance of each node in the network. Census can be used to collect information about the resource usage and performance of each node, which can then be used to detect anomalies and optimize the system.
3. **Distributed load balancing:** In a system where requests are distributed across multiple nodes, census can be used to determine the load on each node and adjust the distribution of requests accordingly. This can help ensure that the system remains responsive and can handle high levels of traffic.
4. **Distributed caching:** In a distributed caching system, census can be used to track the popularity of different objects across the network. This information can then be used to determine which objects to cache and where to store them, in order to optimize performance and reduce network traffic.
5. **Distributed search:** In a distributed search system, census can be used to maintain a distributed index of the data stored across the network. This index can then be used to perform distributed search queries and retrieve relevant results from multiple nodes.
6. **Distributed synchronization:** Census can be used to synchronize clocks across a distributed system, allowing for more precise coordination of tasks.
7. **Consensus algorithms:** Census can be used as a building block in consensus algorithms, such as Paxos and Raft, to reach agreement among distributed nodes. (Extension next pages)
8. **Distributed transaction processing:** Census can be used to track the progress of distributed transactions, ensuring that all nodes involved in the transaction have completed their tasks before committing the transaction.
9. **Fault tolerance:** Census can be used to detect failures in a distributed system, allowing for quick recovery and continued operation even in the presence of failures.
10. **Distributed storage:** Census can be used to keep track of the state of data stored across a distributed system, ensuring that all nodes have access to the most up-to-date version of the data.

9) Census Algorithm - Software Implementation

https://github.com/or1610/Distribued-Software-Systems---Final-Project---Software-Engineering-Part-

YouTube:

https://www.youtube.com/watch?v=BF1pLZASX8s&list=PLZvgWu86XaWkpnQa6-OA7DG6ilM_RnxhW&index=4

10) Stabilization Hierarchy

Self-stabilization is a useful paradigm for forward recovery in distributed systems and networks because it only considers the effect of faults and makes no assumptions about the nature or extent of faults. In practice, when a faulty component is diagnosed in a self-stabilizing network, it can be removed from the system to recover proper behavior automatically.

However, the use of the term "eventually" does not provide any guarantee on the stabilization time, and a single hazard may trigger a correction wave throughout the network. Moreover, the fact that participants in the system may not be able to detect stabilization makes it difficult to have safety-related specifications with self-stabilizing solutions.

To address these limitations, new forms of self-stabilization have been defined, and Figure 11 presents these variants, each denoted by an arrow indicating that it provides weaker guarantees than the next variant.

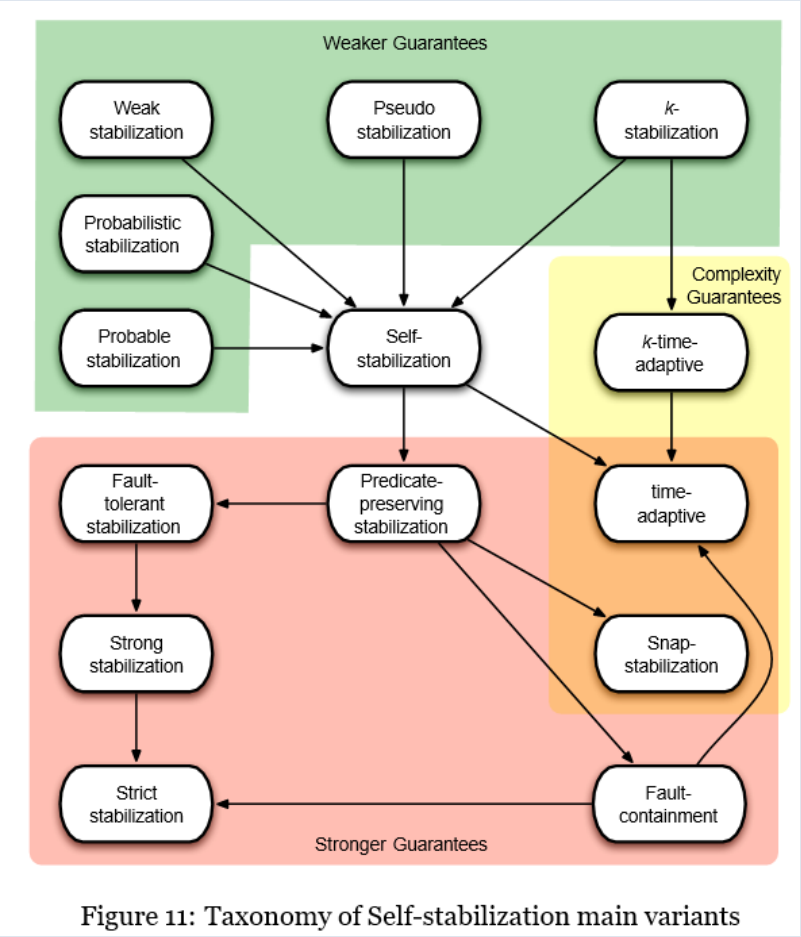


Figure 11: Taxonomy of Self-stabilization main variants

Weaker than self-stabilization

The guarantees that are given are weaker than that of self-stabilization, and this permits in general to solve strictly more difficult specifications than those than can be solved by a self-stabilizing system. Thus, problems that are provably intractable in a strict self-stabilizing setting may be come solvable. This permits to widen the scope of self-stabilization to new applications, while maintaining attractive fault tolerance properties to the developed applications.

1. **Restricting the nature of the faults.**
   This approach, that we denote by probable stabilization consists of considering that truly arbitrary memory corruptions are **highly unlikely**. Probabilistic arguments are used to establish that, in general, memory corruptions that result from faults can be detected using traditional techniques from information theory, such as data redundancy or error detection and correction codes.

2. **Restricting the extent of the faults.**
   (a) ***"visible" stabilization***:
   (b) ***"internal" stabilization***:
   In many studies, only the "visible" stabilization is performed quickly (that is, in time relative to the number of faults that strike the system, rather than in time relative to the size of said system), while the "internal" stabilization most of the time remains proportional to the network's size.

3. **Restricting the stabilization guarantees.**
   (a) ***probabilistic stabilization:*** weakens the convergence requirement by only requiring expected finite time convergence with probability 1.
   (b) ***pseudo-stabilization:*** *removes the guarantee of reaching a configuration.*
   (c) ***Weak stabilization:*** breaks the requirement that every execution reaches a legitimate configuration.

Stronger than self-stabilization

The guarantees that are given are stronger than that of self-stabilization, and this permits in general to solve strictly less difficult specifications than those than can be solved by a self-stabilizing system. That is, the set of problems than can be solved is strictly smaller than the set of problems that can be solved by strictly self-stabilizing algorithms, yet the guarantees are stronger and may even match those of robust algorithms.

1. **Stronger safety guarantees.**
   (a) ***Predicate-preserving stabilization*** refers to the fact that in addition to being self- stabilizing, the algorithm also preserves some distributed predicate on configurations, either in the stabilizing phase or in the stabilized phase, in spite of the occurrence of new faults (of limited nature).
   (b) ***Fault-tolerant self-stabilization:*** characteristic of algorithms that aim at providing tolerance to both arbitrary transient faults (self-stabilization) and permanent ones (robustness), the two main trends in distributed fault-tolerance.
   (c) ***Strict stabilization:*** refers to a different scheme to tolerate both transient and permanent Byzantine faults. The Byzantine contamination radius is defined as the maximum distance from which the effect of Byzantine nodes can be felt. This contamination radius must obviously be as small as possible.

2. **Stronger complexity guarantees:** For certain problems, a memory corruption can cause a cascade of corrections in the entire system, yet it would be natural for the stabilization to be quicker when the number of failures that strike the system is smaller. This is the principle behind time-adaptive self-stabilization, also known as scalable stabilization and fault local stabilization. Note that time adaptive protocols have the property of fault containment in the sense that a visible fault cannot spread on the while network: it is contained near to its initial location before it disappears. It is possible to arrange self-stabilizing, k-stabilizing and time-adaptive algorithms into classes, depending on the difficulty in solving problems that can be solved in

11) Consensus Algorithms – Enrichment

A consensus algorithm is a process in computer science used to achieve agreement on a single data value among distributed processes or systems.

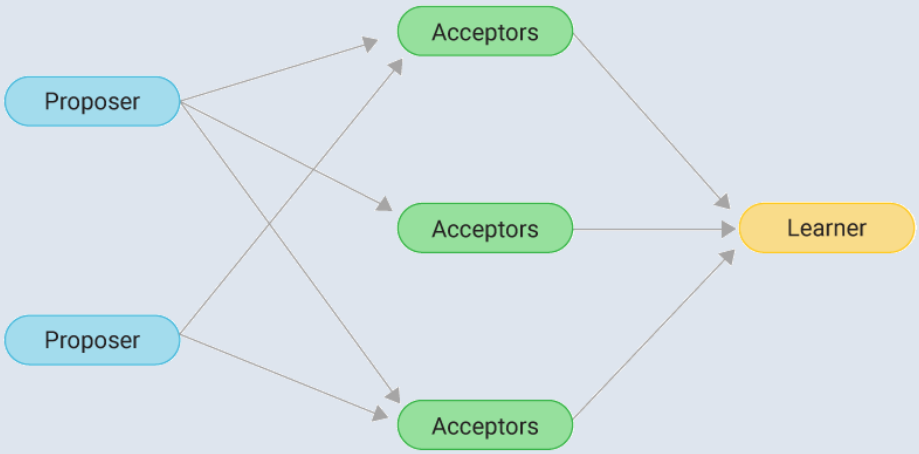
A fundamental problem in distributed computing and multi-agent systems is to achieve overall system reliability in the presence of a few faulty processes.

This often requires coordinating processes to reach consensus or agree on some data value that is needed during computation. Example applications of consensus include agreeing on what transactions to commit to a database in which order, state machine replication, and atomic broadcasts.

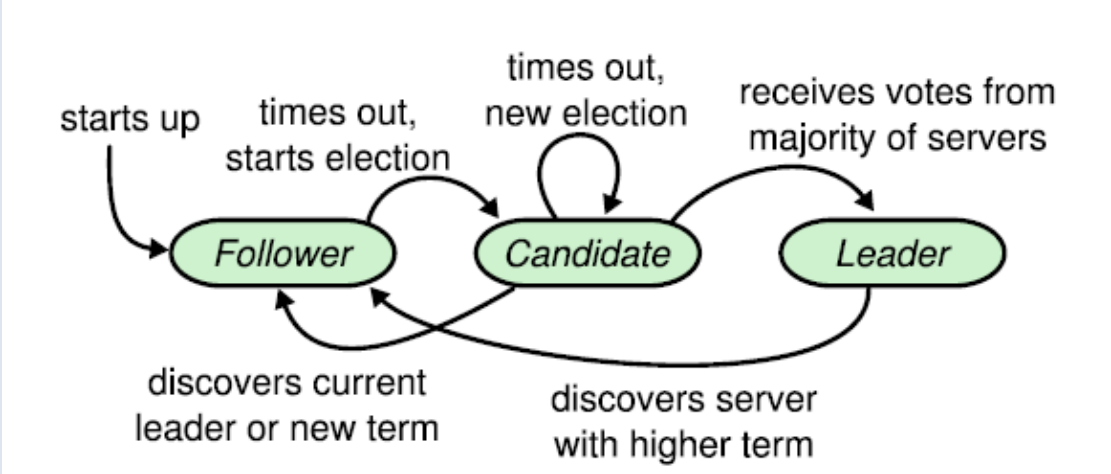
Real-world applications often requiring consensus include cloud computing, clock synchronization, PageRank, opinion formation, smart power grids, state estimation, control of UAVs (and multiple robots/agents in general), load balancing, blockchain, and others.

Here's a brief explanation of three popular consensus algorithms:

1. **Paxos:** Paxos is a consensus algorithm that is commonly used in distributed systems. It is designed to allow a set of nodes to agree on a single value, even in the presence of failures. The algorithm has three phases: prepare, accept, and commit. During the prepare phase, a node proposes a value and tries to get a majority of nodes to agree to it. In the accept phase, nodes agree to a value proposed by the leader, and in the commit phase, the leader commits the agreed-upon value to the network.



2. **Raft:** Raft is another consensus algorithm that is designed to be more understandable than Paxos. It also allows a set of nodes to agree on a single value. Raft uses a leader-follower architecture, where a leader node proposes values and other nodes either agree or disagree. If the leader fails, another node will be elected as the leader



3. **Byzantine fault tolerance (BFT):** BFT is a consensus algorithm that can handle failures and malicious behavior in distributed systems. In BFT, a set of nodes must agree on a value, even if some of the nodes are faulty or malicious. BFT requires a higher number of nodes to agree on a value than Paxos or Raft, which makes it more secure. However, it is also more complex and requires more computational resources.

Byzantine Generals' Problem:

- The Byzantine Generals' Problem (1982 Lamport, Shostak, and Pease)
- Commander defines correct action, lieutenants relay/follow
- n := number of generals
- Trivial for n = 1 or n = 2
- Consider n = 3

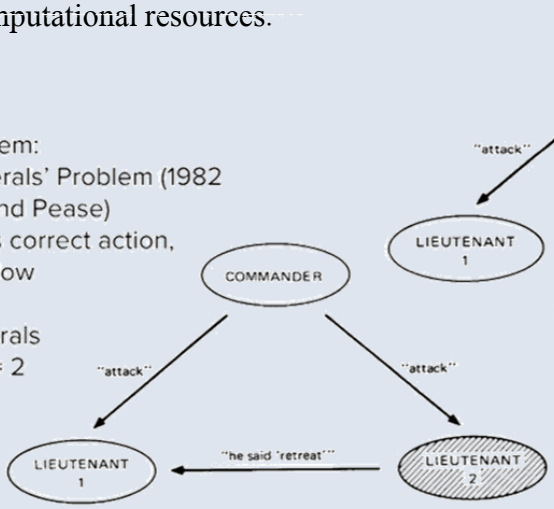


Fig. 1. Lieutenant 2 a traitor.

n = 3, with 1 traitor

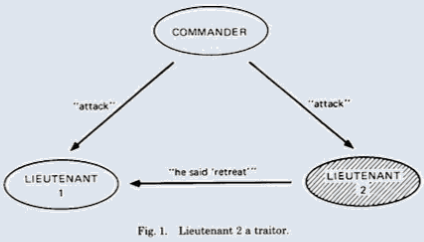


Fig. 1. Lieutenant 2 a traitor.

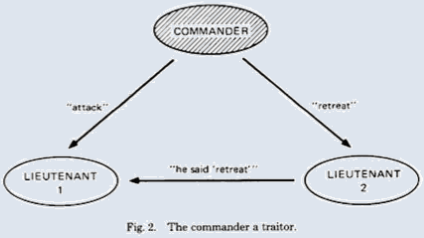


Fig. 2. The commander a traitor.

Assumptions of fault tolerant systems vs Byzantine fault tolerant systems:

- **Fail-stop:** Nodes can crash, not return values, crash detectable by other nodes
- **Byzantine:** Nodes can do all of the above *and* send incorrect/corrupted values, corruption or manipulation harder to detect

12) Project Bibliography

- Tixeuil, S. (January 2010). Self-Stabilizing Algorithms. Sorbonne Université.
- Dolev, S. (January 1990). Self-Stabilization of Dynamic Systems Assuming only Read/Write Atomicity. Ben-Gurion University of the Negev.
- Kohler, S. (2013). Scalable Fault-Containing Self-Stabilization in Dynamic Networks.
- Arora, A., Beauquier, J., Dolev, S., Herman, T., & de Roever, W. P. (Authors). (Ohio State University, Columbus, OH, USA; Université Paris Sud, Orsay, France; Ben Guroin University, Beer-Sheva, Israel; University of Iowa, Iowa City, IA, USA; Universität Kiel, Kiel, Germany). Self-Stabilization.