Chapter 9

Stabilization

In Chapter 6 we learned about systems that are fault-tolerant to partial failures. It was important, however, that a majority of nodes is correct all the time. Moreover, the set of faulty nodes must not change over time.

Can we design a distributed system that survives transient (short-lived) failures, even if *all* nodes are temporarily failing? In other words, can we build a distributed system that *repairs itself*?

9.1 Self-Stabilization

Definition 9.1 (Self-Stabilization)**.** *A distributed system is* self-stabilizing *if, starting from an arbitrary state, it is guaranteed to converge to a legitimate state. If the system is in a legitimate state, it is guaranteed to remain there, provided that no further faults happen. A state is* legitimate *if the state satisfies the specifications of the distributed system.*

Remarks:

- What kind of transient failures can we tolerate? An adversary can crash nodes, or make nodes behave Byzantine. Indeed, temporarily an adversary can do harm in even worse ways, e.g. by corrupting the volatile memory of a node (without the node noticing), or by corrupting messages on the fly (without anybody noticing). However, as all failures are transient, eventually all nodes must work correctly again, that is, crashed nodes get resurrected, Byzantine nodes stop being malicious, messages are being delivered reliably, and the memory of the nodes is secure.
- Clearly, the read only memory (ROM) must be taboo at all times for the adversary. No system can repair itself if the program code itself or constants are corrupted. The adversary can only corrupt the variables in the volatile random access memory (RAM).

Definition 9.2 (Time Complexity)**.** *The time complexity of a self-stabilizing system is the time that passed after the last (transient) failure until the system has converged to a legitimate state again, staying legitimate.*

Remarks:

- Self-stabilization enables a distributed system to recover from a transient fault regardless of its nature. A self-stabilizing system does not have to be initialized as it eventually (after convergence) will behave correctly.
- The self-stabilization property guarantees that the system will end in a correct state after finite (expected) time. This is in contrast to the fault-tolerant algorithms of Chapter 6 that guaranteed that the system is *always* in a correct state.
- Self-stabilization was introduced in a paper by Edsger W. Dijkstra in 1974, in the context of a token ring network. A token ring is an early form of local area network where nodes are arranged in a ring, communicating by a token. The system is correct if there is exactly one token in the ring.
- Let's have a look at one of Dijkstra's simple solutions. Given an oriented ring, we simply call the clockwise neighbor parent (p), and the counter-clockwise neighbor child (c). Also, there is a leader node v_0. Every node v is in a state $S(v) \in \{0, 1, \ldots, n\}$, perpetually informing its child about its state. The token is implicitly passed on by nodes switching state. Upon noticing a change of the parent state $S(p)$, node v executes the following code:

Algorithm 39 Self-stabilizing Token Ring

```
1: if v = v_0 then
2:    if S(v) = S(p) then
3:       S(v) := S(v) + 1 (mod n)
4:    end if
5: else
6:    S(v) := S(p)
7: end if
```

Theorem 9.3. *Algorithm 39 is stabilizes correctly.*

Proof: As long as some nodes or edges are faulty, anything can happen. In self-stabilization, we only consider the system after it is correct (at time t_0, however starting in an arbitrary state).

Every node apart from leader v_0 will always attain the state of its parent. It may happen that one node after the other will learn the current state of the leader. In this case the system stabilizes after the leader increases its state at most n time units after time t_0. It may however be that the leader increases its state even if the system is not stable, e.g. because its parent or parent's parent accidentally had the same state at time t_0.

The leader will increase its state possibly multiple times without reaching stability, however, at some point the leader will reach state s, a state that no other node had at time t_0. (Since there are n nodes and n states, this will eventually happen.) At this point the system must stabilize because the leader cannot push for $s + 1 \pmod n$ until every node (including its parent) has s.

After stabilization, there will always be only one node changing its state, i.e., the system remains in a legitimate state.

□

Remarks:

- For his work Dijkstra received the 2002 ACM PODC Influential Paper Award. Dijkstra passed away shortly after receiving the award. With Dijkstra being such an eminent person in distributed computing (e.g. concurrency, semaphores, mutual exclusion, deadlock, finding shortest paths in graphs, fault-tolerance, self-stabilization), the award was renamed Edsger W. Dijkstra Prize in Distributed Computing.
- Although one might think the time complexity of the algorithm is quite bad, it is asymptotically optimal.
- It can be a lot of fun designing self-stabilizing algorithms. Let us try to build a system, where the nodes organize themselves as a maximal independent set (MIS, Chapter 4):

Algorithm 40 Self-stabilizing MIS

Require: Node IDs

Every node v executes the following code:

1: **do atomically**
2: Join MIS if no neighbor with larger ID joins MIS
3: Send (node ID, MIS or not MIS) to all neighbors
4: **end do**

Remarks:

- Note that the main idea of Algorithm 40 is from Algorithm 16, Chapter 4.
- As long as some nodes are faulty, anything can happen: Faulty nodes may for instance decide to join the MIS, but report to their neighbors that they did not join the MIS. Similarly messages may be corrupted during transport. As soon as the system (nodes, messages) is correct, however, the system will converge to a MIS. (The arguments are the same as in Chapter 4).
- Self-stabilizing algorithms always run in an infinite loop, because transient failures can hit the system at any time. Without the infinite loop, an adversary can always corrupt the solution "after" the algorithm terminated.
- The problem is Algorithm 40 is its time complexity, which may be linear in the number of nodes. This is not very exciting. We need something better! Since Algorithm 40 was just the self-stabilizing variant of the slow MIS Algorithm 16, maybe we can hope to "self-stabilize" some of our fast algorithms from Chapter 4?
- Yes, we can! Indeed there is a general transformation that takes any local algorithm (efficient but not fault-tolerant) and turns it into a self-stabilizing algorithm, keeping the same level of efficiency and efficacy. We present the general transformation below.

Theorem 9.4 (Transformation)**.** *We are given a deterministic local algorithm $\mathcal{A}$ that computes a solution of a given problem in k synchronous communication rounds. Using our transformation, we get a self-stabilizing system with time complexity k. In other words, if the adversary does not corrupt the system for k time units, the solution is stable. In addition, if the adversary does not corrupt any node or message closer than distance k from a node u, node u will be stable.*

Proof: In the proof, we present the transformation. First, however, we need to be more formal about the deterministic local algorithm $\mathcal{A}$. In $\mathcal{A}$, each node of the network computes its decision in k phases. In phase i, node u computes its local variables according to its local variables and received messages of the earlier phases. Then node u sends its messages of phase i to its neighbors. Finally node u receives the messages of phase i from its neighbors. The set of local variables of node u in phase i is given by L_u^i. (In the very first phase, node u initializes its local variables with L_u^1.) The message sent from node u to node v in phase i is denoted by $m_{u,v}^i$. Since the algorithm $\mathcal{A}$ is deterministic, node u can compute its local variables L_u^i and messages $m_{u,*}^i$ of phase i from its state of earlier phases, by simply applying functions f_L and f_m. In particular,

$$L_u^i = f_L(u, L_u^{i-1}, m_{*,u}^{i-1}), \text{ for } i > 1, \text{ and} \tag{9.1}$$

$$m_{u,v}^i = f_m(u, v, L_u^i), \text{ for } i \geq 1. \tag{9.2}$$

The self-stabilizing algorithm needs to simulate all the k phases of the local algorithm $\mathcal{A}$ in parallel. Each node u stores its local variables $L_u^1, \ldots, L_u^k$ as well as all messages received $m_{*,u}^1, \ldots, m_{*,u}^k$ in two tables in RAM. For simplicity, each node u also stores all the sent messages $m_{u,*}^1, \ldots, m_{u,*}^k$ in a third table. If a message or a local variable for a particular phase is unknown, the entry in the table will be marked with a special value $\perp$ ("unknown"). Initially, all entries in the table are $\perp$.

Clearly, in the self-stabilizing model, an adversary can choose to change table values at all times, and even reset these values to $\perp$. Our self-stabilizing algorithm needs to constantly work against this adversary. In particular, each node u runs these two procedures constantly:

- For all neighbors: Send each neighbor v a message containing the complete row of messages of algorithm $\mathcal{A}$, that is, send the vector $(m_{u,v}^1, \ldots, m_{u,v}^k)$ to neighbor v. Similarly, if neighbor u receives such a vector from neighbor v, then neighbor u replaces neighbor v's row in the table of incoming messages by the received vector $(m_{v,u}^1, \ldots, m_{v,u}^k)$.
- Because of the adversary, node u must constantly recompute its local variables (including the initialization) and outgoing message vectors using functions 9.1 and 9.2 respectively.

The proof is by induction. Let $N^i(u)$ be the i-neighborhood of node u (that is, all nodes within distance i of node u). We assume that the adversary has not corrupted any node in $N^k(u)$ since time t_0. At time t_0 all nodes in $N^k(u)$ will check and correct their initialization. Following Equation 9.2, at time t_0 all nodes in $N^k(u)$ will send the correct message entry for the first round $(m_{*,*}^1)$ to all neighbors. Asynchronous messages take at most 1 time unit to be received at

a destination. Hence, using the induction with Equations 9.1 and 9.2 it follows that at time $t_0 + i$, all nodes in $N^{k-i}(u)$ have received the correct messages $m^1_{*,*}, \ldots, m^i_{*,*}$. Consequently, at time $t_0 + k$ node u has received all messages of local algorithm A correctly, and will compute the same result value as in A. □

Remarks:

- Using our transformation (also known as "local checking"), designing self-stabilizing algorithms just turned from art to craft.

- As we have seen, many local algorithms are randomized. This brings two additional problems. Firstly, one may not exactly know how long the algorithm will take. This is not really a problem since we can simply send around the all the messages needed, until the algorithm is finished. The transformation of Theorem 9.4 works also if nodes just send all messages that are not $\perp$. Secondly, we must be careful about the adversary. In particular we need to restrict the adversary such that a node can produce a reproducible sufficiently long string of random bits. This can be achieved by storing the sufficiently long string along with the program code in the read only memory (ROM). Alternatively, the algorithm might not store the random bit string in its ROM, but only the seed for a random bit generator. We need this in order to keep the adversary from reshuffling random bits until the bits become "bad", and the expected (or with high probability) efficacy or efficiency guarantees of the original local algorithm $\mathcal{A}$ cannot be guaranteed anymore.

- Since most local algorithms have only a few communication rounds, and only exchange small messages, the memory overhead of the transformation is usually bearable. In addition, information can often be compressed in a suitable way so that for many algorithms message size will remain polylogarithmic. For example, the information of the fast MIS algorithm (Algorithm 18) can be compressed into a series of random values (one for each round), plus a number and a boolean value. The boolean value represents whether the node joins the MIS, or whether a neighbor of the node joins the MIS. The number tells the round that decision is made. Indeed, the series of random bits can even be compressed just into the random seed value, and the neighbors can compute the random values of each round themselves.

- There is hope that our transformation as well gives good algorithms for mobile networks, that is for networks where the topology of the network may change. Indeed, for deterministic local approximation algorithms, this is true: If the adversary does not change the topology of a node's k-neighborhood in time k, the solution will locally be stable again.

- For randomized local approximation algorithms however, this is not that simple. Assume for example, that we have a randomized local algorithm for the dominating set problem. An adversary can constantly switch the topology of the network, until it finds a topology for which the random bits (which are not really random because these random bits are in ROM) give a solution with a bad approximation ratio. By defining a weaker

adversarial model, we can fix this problem. Essentially, the adversary needs to be oblivious, in the sense that it cannot see the solution. Then it will not be possible for the adversary to restart the random computation if the solution is "too good".

- Self-stabilization is the original approach, and self-organization may be the general theme, but new buzzwords pop up every now and then, e.g. self-configuration, self-management, self-regulation, self-repairing, self-healing, self-optimization, self-adaptivity, or self-protection. Generally all these are summarized as "self-*". One computing giant coined the term "autonomic computing" to reflect the trend of self-managing distributed systems.

9.2 Advanced Stabilization

We finish the chapter with a non-trivial example beyond self-stabilization, showing the beauty and potential of the area: In a small town, every evening each citizen calls all his (or her) friends, asking them whether they will vote for the Democratic or the Republican party at the next election.[1] In our town citizens listen to their friends, and everybody re-chooses his or her affiliation according to the majority of friends.[2] Is this process going to "stabilize" (in one way or another)?

Remarks:

- Is eventually everybody voting for the same party? No.
- Will each citizen eventually stay with the same party? No.
- Will citizens that stayed with the same party for some time, stay with that party forever? No.
- And if their friends also constantly root for the same party? No.
- Will this beast stabilize at all?!? Yes!

Theorem 9.5 (Dems & Reps)**.** *Eventually every citizen is rooting for the same party every other day.*

Proof: To prove that the opinions eventually become fixed or cycle every other day, think of each friendship between citizens as a pair of (directed) edges, one in each direction. Let us say an edge is currently "bad" if the party of the *advising* friend differs from the next-day's party of the *advised* friend. In other words, the edge is bad if the advisor was in the minority. An edge that is not bad, is "good".

Consider the out-edges of citizen c on day t, during which (say) c roots for the Democrats. Assume that during day t, g out-edges of c are good, and b out-edges are bad. Note that $g + b$ is the degree of c. Since g out-edges were good, g friends of c root for the Democrats on day $t+1$. Likewise, b friends of c root for the Republicans on day $t+1$. In other words, on the evening of day $t+1$

[1] We are in the US, and as we know from The Simpsons, you "throw your vote away" if you vote for somebody else. As a consequence our example has two parties only.

[2] Assume for the sake of simplicity that everybody has an odd number of friends.

citizen c will receive g recommendations for Democrats, and b for Republicans. We distinguish two cases:

- $g > b$: In this case, citizen c will still (or again) root for the Democrats on day $t + 2$. Note that in this case, on day $t + 1$, exactly g in-edges of c are good, and exactly b in-edges are bad. In other words, the number of bad out-edges on day t is exactly the number of bad in-edges on day $t + 1$.
- $g < b$: In this case, citizen c will root for the Republicans on day $t + 2$. Note that in this case, on day $t + 1$, exactly b in-edges of c are good, and exactly g in-edges are bad. In other words, the number of bad out-edges on day t was exactly the number of good in-edges on day $t + 1$ (and vice versa). Since citizen c is rooting for the Republicans, the number of bad out-edges on day t was strictly larger than the number of bad in-edges on day $t + 1$.

Every edge is an out-edge on day t, and an in-edge on day $t + 1$. Since in both cases the number of bad edges do not increase, the total number of bad edges B cannot increase. In fact, if any node switches its party from day t to $t + 2$, we know that the total number of bad edges strictly decreases. But B cannot decrease forever. Once B hits it minimum, the system stabilizes in the sense that every citizen will either stick with his or her party forever or flip-flop every day – the system "stabilizes". □

Remarks:

- The model can be generalized considerably by, for example, adding weights to vertices (meaning some citizens' opinions are more important than others), allowing loops (citizens who consider their own current opinions as well), allowing tie-breaking mechanisms, and even allowing different thresholds for party changes.
- How long does it takes until the system stabilizes?
- Some of you may be reminded of Conway's Game of Life: We are given an infinite two-dimensional grid of cells, each of which is in one of two possible states, *dead* or *alive*. Every cell interacts with its eight neighbors. In each round, the following transitions occur: Any live cell with fewer than two live neighbors dies, as if caused by lonelyness. Any live cell with more than three live neighbors dies, as if by overcrowding. Any live cell with two or three live neighbors lives on to the next generation. Any dead cell with exactly three live neighbors is "born" and becomes a live cell. The initial pattern constitutes the "seed" of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed, births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations. John Conway figured that these rules were enough to generate interesting situations, including "breeders" with create "guns" which in turn create "gliders". As such Life in some sense answers an old question by John von Neumann, whether there can be a simple machine that can build copies of itself. In fact Life is Turing complete, that is, as powerful as any computer.

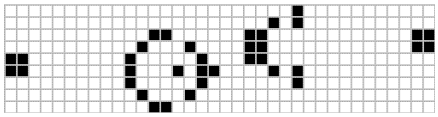

Figure 9.1: A "glider gun"...

Figure 9.2: ...in action.