Lecture 9

Self-Stabilization and Recovery

Previously, we have seen ways of handling permanent faults: nodes crashed and never booted up again, or they turned Byzantine, which was an untreatable condition. However, not every job needs the same tool. For instance, Byzantine tolerance needs that $f < n/3$ nodes are faulty, and that might not be true over extended periods of time. Instead, nodes or communication links (i.e., edges) may undergo *transient* failures. They recover eventually, but that doesn't mean the state of their volatile memory hasn't been corrupted!

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?*

Definition 9.1 (Self-Stabilization)**.** *Denote by $\mathcal{S}$ the* state space *of a distributed system, i.e., the product space over all individual nodes' state spaces. An* execution *of an algorithm is a (bounded or unbounded) sequence of state transitions that is valid in that it follows the rules given by the algorithm and the execution model. Define a set $\mathcal{C}$ of* correct executions*, i.e., executions that satisfy some desirable properties. The system/algorithm is called* self-stabilizing *(w.r.t. $\mathcal{C}$) iff every possible execution, no matter what the initial state, has a correct suffix.*

Usually, the information that the algorithm stabilizes is of limited use, as the time for it to do so might be unbounded. This means that typically we are interested in algorithms that have a small *stabilization time*. The definition of self-stabilization given above is very general, so there wasn't any notion of time. It needs to be derived from the execution model, which we didn't specify! So let's define what we mean for synchronous and asynchronous message passing systems.

Definition 9.2 (Stabilization time)**.** *Fix a set of parameters, e.g., the number of nodes n and the diameter of the network D. The* stabilization time *of a synchronous self-stabilizing system is the maximum number of rounds R so that removing the first R rounds from the execution results in a correct execution, taken over all executions on n-node graphs of diameter D. If this maximum doesn't exist, the algorithm has* unbounded *stabilization time. For an asynchronous system, we do the same with respect to asynchronous rounds, i.e., after*

normalizing the maximum message delay to 1, we check the maximum number of time units we need to cut off of the front of an execution to make the suffix correct.

Remarks:

- $\mathcal{S}$ needs not be complicated. For instance, for the MIS algorithm from the lecture, it was merely a few bits: Is the node in the MIS? Is it terminated? Are we in a round for trying to join the MIS or just propagating the information which nodes did? Etc. Self-stabilization then means that even if who's in the MIS and who's not is completely messed up, the algorithm will re-construct an MIS.
- This definition is quite abstract, but can be applied to essentially any state-based description of a system. Of course, $\mathcal{C}$ should be a set of executions in which the system is considered to behave "correctly."
- The requirement here is very strong. Ending up in $\mathcal{C}$ from *any* state is equivalent to saying that *anything* can happen to the volatile memory (a.k.a. states) during a period of transient failures.
- The program code of the nodes and their hardware capabilities must not be changed. Also, the communication infrastructure must operate correctly after transient faults ceased. Otherwise there's no way of guaranteeing that the nodes behave as intended by the algorithm, and we can't hope to guarantee recovery!
- This seems obvious, but it means that one has to be careful. What if a simple bit flip tells the node to stop running the algorithm?
- Some things can't be self-stabilizing. For instance, the (single-shot) consensus algorithms we discussed generates some output based on the inputs. There is no way that we can ensure that everything works out if we make everyone believe the inputs were actually different! Let's try to capture this issue better.

9.1 Self-stabilizing Algorithms can't Terminate

Lemma 9.3. *A self-stabilizing algorithm can never terminate, unless for each node there is a single output that is always correct.*

Proof. Suppose there are two possible conflicting outputs for a node v. More precisely, there are system/input combinations for which the algorithm must output different values. Any terminating algorithm we can let run until it terminates. Then we switch the system/input combination, but set the state of some node w to the state of v from above (a transient fault may cause this). We have found an execution in which w doesn't change its state any more (as it has terminated), but has incorrect output. Hence, the execution has no correct suffix, i.e., the algorithm can't be self-stabilizing! □

Remarks:

- For instance, the only self-stabilizing coloring algorithm that terminates assigns a unique color to each node identifier – e.g., the identifier itself. Boring!
- That's also why consensus doesn't make a lot of sense here, at least not in the usual way the problem is posed.
- However, having inputs *does* make sense. One can give the inputs in registers that cannot be touched and require the algorithm to stabilize to correct outputs for these inputs. If the input registers are affected by a fault, it's also okay that the outputs change accordingly.

9.2 Dijkstra's Token Ring

One of the first self-stabilizing algorithms was Dijkstra's token ring network. A token ring is an early form of a local area network where nodes are arranged in a directed ring, communicating by a token. The system is correct if there is exactly one token in the ring, which keeps being passed around. Let's have a look at a simple solution. Given an oriented ring, we simply call the clockwise neighbor *child c,* and the counterclockwise neighbor *parent p.* Also, there is a leader node v_0. Every node v is in a state $S(v) \in \{0, 1, \ldots, n-1\}$, 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 19 Self-stabilizing token ring. A node with $S(v) \neq S(p)$ holds a token – except for v_0, which has the token if $S(v) = S(p)$. In each round, each node announces its state to its child, which then updates its own state as below.

```
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 9.4. *In a synchronous system, Algorithm 19 is self-stabilizing with stabilization time at most* $3n$.

Proof. Suppose the leader is in a state s that nobody else has in some round. By the rules of the algorithm, the state propagates around the ring, and exactly n rounds later the leader switches to state $(s+1) \bmod n$; at this time, all other nodes are in state s. From that point on, the algorithm circulates a single token around the ring.

Now assume that for n rounds, the leader does *not* attain a state that nobody else had initially. Within n rounds, the original state of the leader propagates around the ring, reaching the leader again. Any state that is not eliminated during that time must be attained by the leader, which increases its state by

1 mod n each time it switches its state. Note that the leader cannot perform this operation n times without reaching a state that no node in the system initially had. Therefore, after n rounds the leader is in a state s such that state $s+1 \bmod n$ is not present in the system. At most n rounds later, the leader will increase its state again, implying stabilization within a total of $3n$ rounds. □

Remarks:

- This is asymptotically optimal, as two tokens could be on opposite sides of the ring. It takes at least $n/2$ rounds to distinguish this from a 1-token system.
- The algorithm also works in asynchronous systems. Can you see how to generalize the above proof?
- We defined the stabilization time as a specific value. Usually determining it precisely is very hard (or just tedious), so we tend to give upper bounds on the stabilization time. Language then deteriorates a bit and we tend to talk of an algorithm "having stabilization time S" when actually meaning that it has stabilization time at most S.
- It can be a lot of fun designing self-stabilizing algorithms, and it's not always as difficult as one might expect. However, it's essential to do baby steps. Initially, one shows seemingly very weak statements, but the resulting increased degree of organization in the system makes it much easier to follow up with stronger properties. The above proof exemplifies this; reversing the order helped, as it made clear why we wanted to show the intermediate claim that eventually the leader attains a state that wasn't present before.

9.3 Synchronous = Self-stabilizing Asynchronous!

Finding and proving correct self-stabilizing algorithms can be difficult. Let's automate the process!

We want to transform an arbitrary synchronous R-round message-passing algorithm $\mathcal{A}$ into a self-stabilizing asynchronous one. This way we can reason about things in a simplified setting (synchronous message-passing), immediately obtain a result in the asynchronous model, and self-stabilization comes for free. The hard part is to make sure that all nodes think they are in the same round of $\mathcal{A}$; instead of solving this problem, though, each node v simulates R copies $v_1, \ldots, v_R$ of itself, where v_i is in round i. Each round, v simply communicates all messages all v_i send as a bundle, and uses the received message bundles in the next round.

Theorem 9.5. *Given a deterministic synchronous message-passing algorithm $\mathcal{A}$ that runs for R rounds, we can construct an asynchronous self-stabilizing algorithm $T(\mathcal{A})$ that stabilizes in R time. If the message size of $\mathcal{A}$ is at most M_i in round i, the message size of $T(\mathcal{A})$ is $\sum_{i=1}^{R} M_i$.*

Proof. For each round i of $\mathcal{A}$, each node $v \in V$ simulates a copy v_i of itself in round i. Each copy sends messages to some neighbors w_i, $\{v, w\} \in V$, and receives the messages of them. Then it computes its state for round $i+1$, i.e., the state of v_{i+1}. The state of v_1 is determined solely by the input of v, and from the state of v_R the output of v can be determined. For this simulation, v needs to keep communicating the messages of all copies $v_1, \ldots, v_R$, yielding message size $\sum_{i=1}^{R} M_i$.

We know from Lemma 9.3 that a self-stabilizing algorithm cannot terminate. Hence, v will just keep sending the message vectors to all neighbors and updating the state of each of its copies v_i whenever receiving a message vector from a neighbor. Now why is the algorithm self-stabilizing? Well, we know that once transient faults cease, each node will correctly compute the messages of the first round and send them in the message vectors. Once a node received such vectors from all neighbors, it locally simulates round 2 correctly and sends message vectors with correct messages for rounds 1 and 2. This takes at most one time unit in terms of asynchronous time complexity. By induction, after R time units the correct results on termination are determined by all nodes. □

Remarks:

- Using this transformation, also known as *local checking*, designing self-stabilizing algorithms just turned from art to craft.
- Note that if R is a function of, e.g., n, (an upper bound on) n must be known and hardwired into the algorithm.
- The asynchronous model needs to be slightly augmented here to make sense. Transmitting "continuously" in the message passing model would mean to send an infinite number of messages. Instead, one assumes that nodes can perform a "busy-wait," for which they keep executing a loop. The next loop execution is one of the possible actions the scheduler can trigger (apart from receiving a message).
- Note that a node may send a huge number of messages during the stabilization phase. However, in principle it's ok to mitigate this problem by nodes waiting and collecting messages for some time before processing them.
- In the asynchronous shared memory model this problem doesn't exist, simply because the busy-wait is already a necessary part of the model.
- This transformation does not work for randomized algorithms. Execution and output would change all the time.
- In practice, one simply uses "pseudo-randomness." We fix sufficiently long random bit strings in advance, giving them to the nodes as part of the program memory.
- Of course the result is – technically – a deterministic algorithm, and all respective restrictions apply. One can always find a bad input/schedule combination, unless guarantees are, in fact, deterministic. However, as soon as the adversary is "oblivious," i.e., needs to make its decisions independently of the pseudo-random strings, we're good again. This usually is fine, unless there is an *actual* adversary who exploits this weakness!

- The impact on the message size is small for very fast algorithms.
- If there is a local fault or the topology or inputs change locally, this will only cause corrections in an R-hop neighborhood of the faults for an R-round algorithm.
- Ergo, this transformation rocks when applied to very fast algorithms!

9.4 Non-local Recovery

Recall that there are global problems, like MST computation. Using the above transformation would yield prohibitively large messages, even when starting from an algorithm using small messages! One way to handle this would be to keep executing a respective algorithm repeatedly, controlling it via a BFS tree. However, this would imply large stabilization times.

Can we have something simpler? If everything can be messed up, the answer is no. The lower bounds tell us that we can't be faster in the worst case. But if there are only few changes, we can exploit the convenient structure of the MST problem again.

Lemma 9.6. *If at most k edges in a (connected) weighted graph change their weight, appear, or are deleted (such that the graph is still connected), the new MST differs in at most k edges from the previous one.*

Proof. Consider a single edge e and suppose it increases its weight or gets deleted. If it's not in the MST M, nothing changes. Otherwise, it might not be in the MST any more. Denote by e' the lightest edge that does not close a cycle with $M \setminus \{e\}$. Then $(M \setminus \{e\}) \cup \{e'\}$ is the new MST: Running Kruskal on the new graph will select all edges from $E \cap M$ that are lighter than e', then e', and then (as the connectivity components at this point are identical to the run on the original graph) the remaining edges of $M \setminus \{e\}$.

Now suppose an edge e appears or decreases its weight. If it already was in the MST M or is not in the new MST, nothing changes. Otherwise, there is a unique edge $e' \in M$ heaviest in a cycle in $M \cup \{e\}$. The same reasoning as above shows that the new MST is $(M \setminus \{e'\}) \cup \{e\}$.

This shows the claim for $k = 1$. By induction on k, it follows for any k. □

This yields a very easy way of fixing a "broken" MST depending on the number of changes.

Theorem 9.7. *Assume that we have a fixed BFS tree that does not change. There is an MST algorithm that can recover from k changes in edge weights, deletions, or insertions in $\mathcal{O}(k + D)$ rounds, assuming that it had terminated before.*

Proof sketch. Run the distributed version of Kruskal on the BFS tree until termination. Now suppose k changes are made. The nodes noticing these will start sending corrections to their parents concerning their local forests (i.e., report insertions, deletions, and weight changes). As Lemma 9.6 generalizes to the min-weight maximal forests maintained locally by the BFS tree nodes in the distributed version of Kruskal's algorithm, none of these forests change by more than k edge "swaps." Hence, each node needs to send at most k messages

to report updates to its parent. They may send some intermediate incorrect values, but the pipelining argument generalizes to this setting, too. Hence the root will learn about the new MST within $\mathcal{O}(k + D)$ rounds. □

Remarks:

- This demonstrates how important insights into the structure of problems are. This idea is very straightforward once the distributed version of Kruskal's algorithm is known, which relies on the matroid structure of the problem. This is another way of exploiting this structure!
- As pointed out, the lower bounds imply that a self-stabilizing algorithm cannot be that fast. The issue is that the entire computation needs to be repeated, as any pre-computed information cannot be trusted!
- Handling changes in the BFS tree is more involved. While adding or deleting a single edge can also only change a single BFS tree edge (just apply Lemma 9.6 with all edges having weight 1), adapting the distributed data structure given by the states of the nodes becomes a nuisance.

9.5 2-Party Systems Stabilize

We finish the chapter with another 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)?

- 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 9.8 (Dems & Reps)**.** *Eventually every citizen is rooting for the same party every other day.*[3]

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,

[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.

[3] Hence the term "swing voter."

the edge is bad if the advised friend did not follow the advisor's opinion (which means that 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 $t+1$ 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 $t + 1$.

We account for every edge as out-edge on day t, and as in-edge on day $t + 1$. Since in both of the above cases the number of bad edges does 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 its 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), adding weights to edges (meaning the influence between some citizens is stronger than between others), allowing loops (citizens who consider their own current opinions as well), allowing tie-breaking mechanisms, and even allowing different thresholds for party changes.

- 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 loneliness. 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" which 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.

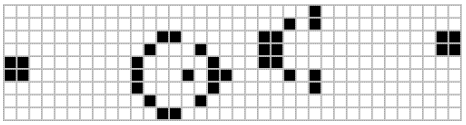

Figure 9.1: A "glider gun"...

Figure 9.2: ... in action.

Bibliographic Notes

Self-stabilization was first introduced in a paper by Edsger W. Dijkstra in 1974 [Dij74], in the context of a token ring network. It was shown that the ring stabilizes in time $\Theta(n)$. For his work Dijkstra received the 2002 ACM PODC Influential Paper Award. Shortly after receiving the award he passed away. 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. In 1991 Awerbuch et al. showed that any algorithm can be modified into a self-stabilizing algorithm that stabilizes in the same time that is needed to compute the solution from scratch [APSV91]; this construction is explained in Theorem 9.5.

For fast "distributed correction" of MSTs see the work by Peleg [Pel98]. This paper in fact deals with the more general case of matroids, and hence spotlights how the matroid structure is exploited by the distributed version of Kruskal's algorithm as well as the adaption to changing edge sets and weights.

The Republicans vs. Democrats problem was popularized by Peter Winkler, in his column "Puzzled" [Win08]. Goles et al. already proved in [GO80] that any configuration of any such system with symmetric edge weights will end up in a situation where each citizen votes for the same party every second day. The understanding of this problem has been significantly extended recently [FKW13, KPW14]. Closely related to this puzzle is the well known Game of Life which was described by the mathematician John Conway and made popular by Martin Gardner [Gar70].

This lecture is in wide parts based on material by Roger Wattenhofer. Thanks!

Bibliography

[APSV91] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-Stabilization By Local Checking and Correction. In *In Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, 1991.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):943–644, November 1974.

[FKW13] Silvio Frischknecht, Barbara Keller, and Roger Wattenhofer. Convergence in (Social) Influence Networks. In *27th International Symposium on Distributed Computing (DISC), Jerusalem, Israel*, October 2013.

[Gar70] M. Gardner. Mathematical Games: The fantastic combinations of John Conway's new solitaire game Life. *Scientific American*, 223:120–123, October 1970.

[GO80] E. Goles and J. Olivos. Periodic behavior of generalized threshold functions. *Discrete Mathematics*, 30:187–189, 1980.

[KPW14] Barbara Keller, David Peleg, and Roger Wattenhofer. How even Tiny Influence can have a Big Impact! In *7th International Conference on Fun with Algorithms (FUN), Lipari Island, Italy*, July 2014.

[Pel98] David Peleg. Distributed Matroid Basis Completion via Elimination Upcast and Distributed Correction of Minimum-Weight Spanning Trees. In *Proc. 25th Colloquium on Automata, Languages and Programming (ICALP)*, pages 164–175, 1998.

[Win08] P. Winkler. Puzzled. *Communications of the ACM*, 51(9):103–103, August 2008.