Self Stabilization (Part I)

Papers

- Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. Communication of the ACM, 1974.
- George Varghese. Self-stabilization by counter flushing. SIAM Journal on Computing, 30(2):486-510, 2000. Society for Industrial and Applied Mathematics
- Adam M. Costello and George Varghese. Self-stabilization by window washing.Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC '96), Philadelphia, Pa., USA, pages 35-44, May 1996.

1 Dijkstra's paper

Dijkstra starts by motivating synchronization with *legitimate state.* Then he talks about self-stabilization as a natural requirement for distributed system. He talks in passing about two motivating scenarios ("outside the scope of this paper"): initialization and fault tolerance.

Model: read-modify-write (aka *guarded commands*), scheduling done by a *central daemon.* "Privilege" is that a guard is enabled.

Legitimate **state**: 1. at least one privilege (*no deadlock*) 2. *closure* of the set *Non-triviality*: (3) there is at least one state for each privilege, and (4) any legal is reachable from any legal.

self-stabilization requirement:
For all initial states and for all schedules: eventually legal state (*convergence*).

Note that the requirement that there is always a privilege is just to rule out trivialities: otherwise, could start the system in a deadlocked state, and then schedule may not be well-defined.

He argues that this formulation alone makes it unclear whether there exists a self-stabilizing solution (now we know better). I find it hard to understand why should we care at this point...

1.1 The problem

The actual problem is for a ring of processes, and the legitimate states are those with exactly one privilege. This is actually called *token passing on a ring*, or *mutual exclusion.* Interesting to note that the mutual exclusion problem is solved *assuming* mutual exclusion primitive: the daemon!

Dijkstra does not give any intuition or proofs: "appreciation is left as an exercise to the reader" (!) It will be 12 years until he publishes correctness proofs for the three algorithms (first issue of *Distributed Computing*, 1986).

All three solutions use a distinguished node with a different code (sometimes two, called *top* and *bottom*). Dijkstra mentions that there is no uniform solution. It has been shown [Burns-Pachl] that if the size of the ring is prime, then there is a solution (they also discuss the *distributed daemon*).

1.2 Solution 1

$N+1$ processes numbered $0, \ldots, N$. Each node i has a counter C_i in the range $\{0, \ldots, K-1\}$ for $K \geq N+1$.

Process 0: if $C_0 = C_N$ then $C_0 \leftarrow (C_0 + 1) \bmod K$. Process i, for $i > 0$: if $C_i \neq C_{i-1}$ then $C_i \leftarrow C_{i-1}$.

Legal states: j smallest-index processes with value ℓ (j may be 0), and $N+1-j$ process with value $(\ell - 1) \bmod K$. If $j > 0$, only process has privilege. Otherwise, Process 0 has privilege.

Legal Executions: privilege rotates in the cycle from i to $(i+1) \bmod K$.

Proof of stabilization:

1. regardless of the initial state, eventually process 0 increments its counter. This is because other processes only copy. Formally: for each state, write a vector of N bits with bit i equal 1 iff $C_i \neq C_{i-1}$. Each move of a process $i > 0$ strictly decreases this vector lexicographically. When the vector is all zeros, all processes have the same value and only process 0 can move. 2. regardless of the initial state, eventually process 0 will have a unique value. This is because only 0 creates new values, and by Lemma 1 it will repeatedly do so. 3. Starting from a state where 0 has a unique value m, there eventually be a state where all process have the value m. Proof is like for Lemma 1, with the added observation that process 0 will not move until *all* values are equal to m.

The closure property can be seen by induction–will not go into that.

1.3 Solution 2

This solution is trickier: only 2 bits per process called u and x. Intuition: $u_i = 1$ iff there's a privilege in $j > i$. x determines the direction of the token movement.

Code: now the code for process N is different too.

Process 0: if $x_0 = x_1$ and $up_1 = 0$ then $x_0 \leftarrow 1 - x_0$. Process N: if $x_N \neq x_{N-1}$ then $x_N \leftarrow x_{N-1}$. $u_N = 0$ always. Process i for $0 < i < N$: if $x_i \neq x_{i-1}$ then $x_i \leftarrow x_{i-1}$; $u_i \leftarrow 1$; // move up else if $u_i = 1$ and $u_{i+1} = 0$ then $u_i \leftarrow 0$ // move down

Note that processes 0 and N can be combined into one 4-state process.

Legal executions: start with all bits zero. Process 0 flips x_0 to 1. Then processes i flip their x_i to 1 and u_i to 1 (moving up) Eventually process N flips x_N, and then Processes i flips their u_i bits (move down). Now this returns with the x bits reversed.

Moving up copies the x from below and sets u value. Moving down

Legal states: there exists some j such that * all processes $0..j$ have the same x and u value; * all processes $j+1..N$ have opposite u value, and they have the same x value, that may be the same as the x value of the lower processes. If the x values differ, the token is going up, otherwise it is going down.

Will not go into proof of correctness today–maybe next time.

Instead– we'll deal today with generalizations of the first algorithm.

Before that: a digression.

1.4 Why care about the size of the state?

In many cases, distributed algorithms use unbounded sequence numbers. Theoreticians complain, by why should we really care?
Answer: Usually we shouldn't! 64-bit counters are good for any practical purpose: we will never see a wrap-around, and correctness will be maintained.
— except for self-stabilization... where the initial state is arbitrary.
The point is that in the arbitrary initial state, the counter value could be the maximum possible, making the next increment to wrap-around, thus usually killing the correctness.
Moral: never by an unbounded-solution for self-stabilization. It makes no sense!

1.5 Historical perspective

Dijkstra's paper went largely unnoticed (Kruijer, his student, published a generalization in 1979). Only in 1983 Lamport, in a keynote address to PODC, pointed out the significance of self-stabilization to fault-tolerance. (He also devised a few self-stabilizing algorithms, e.g., for mutual exclusion.) The impact was staggering: around the late 80's–early 90's a significant part of the distributed computing community was consumed with self-stabilizing algorithms (Recent bibliography: more than 500 papers).

So why is self-stabilization that interesting?

It abstracts all *transient faults*: faults that stop affecting the system at some point. When they go, we are left with an *arbitrary initial state*! If the system can deal with that, it can deal with any transient fault.

Compare: Byzantine faults restrict number of affected processes. Crash faults, omission: restrict nature of faults. Self-stabilization restricts their *time-span*.

Also: automatic initialization (even without faults) is highly desirable in any distributed system.

The question is what is the price one pays to get this kind of resilience.

2 Counter Flushing

Switch model to message passing: more realistic.

Cannot have usual interpretation of links: if links can hold an unbounded number of messages, then one cannot guarantee bounded stabilization time. Settle for Bounded Data Link: at most L_{max} outstanding packets in any given time. Time complexity: an action which is continuously enabled is taken within 1 time unit. Message delivery time is 1 time unit.

Maintain the distinguished leader assumption.

Avoiding Deadlocks: periodic retransmission of states.

2.1 Rings

Token passing: same algorithm as before. Must have the counter be at least c_{max}, the maximal number of counters that can be concurrently in the system. For BDL, c_{max} is about $|E|L_{max}+O(V)$

(depending on how many copies of the counter are stored in each node). time complexity: Easy to see 3 round-trips ($O(n)$ time): one round-trip for each Lemma above. (Truth: 2 round trips.)

Applications: ordered broadcasts (request-response). Each request is matched with a response by number. End-to-end communication: requests are data sends, responses are acks.

2.2 Trees

Basic task: PIF (Propagation of Information with Feedback). Extension of the request-response task to trees: The tree is rooted at the leader. Leader initiates a broadcast; each node that gets the message from its parent propagates to children and waits; leaves return ack immediately; when a node gets all acks from children, it forwards an ack to parent; when root relieved all acks, it can start a new cycle. Each relaying of the message can do some associative processing (eg, sum, max).

Code: Each node has an array with an entry for each child, indicating whether an ACK is expected and a counter showing the current phase number

initiation: root increments its counter a node continuously copies counter from parent acks are sent with counter value, setting the entry in the ack_expected array

Legal state: * The "upper portion" of the tree (including messages) has equal counter value * If the root is not expecting any ACK then the upper portion consists of the whole tree * if a packet going to parent with value like root, then the child is done and has the same value. * if no ack expected from a child and the parent has same value as root, then child is done, and all values on link from child to parent have the root value.

Stabilization proof: an extension of the ring proof. 1. After $O(h)$ time, the leader will choose another value. Otherwise: after h time all leaves will have the the leader's value, and after additional j time units all the leader's children will have that value too. 2. After $O(h)$ additional time, leader will choose a fresh value. 3. After $O(h)$ additional time, the system will be in a legal state.

Extension to general graphs: Using a spanning tree protocol

Bellman-Ford is easily self-stabilizing if we have a fixed leader).

Source: $d \leftarrow 0$ Other nodes: $d_v \leftarrow 1 + \min\{d_u \mid u \text{ is a neighbor of } v\}$.

Proof sketch for the synchronous model:

Lemma 1: After i time units, for any node v, $d_v \geq \min(i, distance(s, v))$ Proof: by induction on i. Base: $i = 0$. Step: Assume that $d_u \geq \min(i, distance(s, u))$. Consider v: If it assigns $d > i + 1$, we are done. Otherwise, $d < i + 1$. In this case v has a neighbor u with $d_u = d - 1$ in the previous step, and by induction $distance(s, u) \leq d - 1$, hence $distance(s, v) \leq d$. QED

Lemma 2: At time t, if $distance(s, v) \leq t$ then $d_v \leq distance(s, v)$. Proof: By induction on the distances. Base: $t = 0$, node is the source, trivial. Step: Suppose lemma holds for all nodes at distance d and all times after t. Consider a node v at distance $d + 1$: it has a neighbor at distance d, which by induction has its variable to be at most d by time d, hence d_v is set to at most d+1 at any time from $t + 1$ and on. QED

3 Window washing

While counter flushing extended the basic idea to message-passing systems and trees, window washing extends the idea to *pipelining*.

Observation: In the PIF protocol, a new round can start only after the previous one has *ended*. This means that if we have k jobs to do, and the cross-system latency is d, then it will take us $O(kd)$ to finish is all. If we could start a job *before* its predecessor is done, then one can hope to get overall complexity of $O(k+d)$ time, which can be significant if $d \gg 1$.

In the window washing paper, Varghese shows how to make the pipeline (sliding window) self-stabilizing.

3.1 Basic scenario: 1 sender, 1 receiver

non-stabilizing code: Sndr: if $seq \in [L+1, L+w]$ send seq Rcvr: if $seq = R+1$ then $R \leftarrow R+1$ and deliver message Rcvr: send $ack(R)$ Sndr: if rcvd $ack(R)$ $L \leftarrow R$.

sender maintains the next msg to be sent seq, last ack received L. if next msg to be sent not larger than ack+W, it is sent. When ack arrives, last ack is bumped up.

receiver maintain last seq to accept R. If next arrives-accepts. occasionally send ack on last accepted.

does not stabilize!

Example: modulus = 20; window = 3; c_{max}: 7 values. sender = 12 10; 7; reliever = 4; 1; 18; 15 apply transitions in order, get a cyclic shift. can persist forever!

Making stabilizing: lesson from counter flushing: make modulus more than window size times c_{max}.

More: * sender ignores acks not in $[L+1, L+w]$ * sender sends L too, and the reliever copies it if its current R is not in the window.

Proof idea: Note that the sender increases its L value by at most w for each ack. It follows that until the sender gets the first genuine ack (generated by the first genuine message the receiver got), the sender can advance its L value by at most wc_{max}. Thereafter, L does not change until an ack in $[L+1, L+w]$ arrives. Hence no new sequence numbers will enter the system. Eventually, all acks with different values will vanish, and the system will stabilize.

3.2 Extension: Tree

The root is the only sender, all nodes are receivers. Essentially the idea is to maintain sender-receiver pair state for each receiver. Messages are propagated to children, acks are collected and sent to parent. Sender advances according to the smallest ack received. Separation between the data tree and ack tree. proposal for a constant-size messages: send the *range* of acks, so long as it's no more than $MaxWidth \leq M/2$, where M is the modulus. If the range is larger, send $WIDE$. Arithmetic is as expected. $Wide$ values are ignored by the sender.