Dijkstra's Protocols as Examples of Local Checking and Counter Flushing

George Varghese*

July 11, 1994

1 Introduction

We will focus on the ability of network protocols to stabilize to "correct behavior" after *arbitrary* initial perturbation. This property was called *self-stabilization* by Dijkstra [Dij74]. The "self" emphasizes the ability of the system to stabilize by *itself* without manual intervention.

The purpose of this note is to introduce two general techniques for making protocols self-stabilizing. We will do so using two examples from the seminal paper of Dijkstra [Dij74].

2 Definitions

All the existing definitions of stabilization are in terms of the states and executions of a system. We will use a definition of stabilization that corresponds to the standard definitions (for example, that of Katz and Perry [KP90]). However, we will use the I/O automaton model of Lynch and Tuttle ([LT89]) as our modeling tool.

*Washington University, St. Louis, work done while at Lab. for Computer Science, MIT. These notes are based on a lecture given in the Distributed Algorithms Course (6.852J) at MIT in Dec 1992

The I/O Automaton model models a protocol as an automaton (state machine) with a few extra features. First, transitions of the automaton are labeled with action names, and all actions are classified into three categories: input, output, and internal. Intuitively, *input actions* are actions caused by the external world or environment and to which the automaton must respond. *Output actions* are actions caused by the automaton and to which the environment must respond. Finally, *internal actions* are transitions that are neither input or output actions: such actions only change the internal state of the automaton.

Recall also that when an automaton "runs" it produces an *execution*, an alternating sequence of states and actions that represent a sequence of state transitions. The external behavior corresponding to an execution a is the subsequence of a consisting only of input and output actions.

Suppose we define the correctness of an automaton in terms of a set C of legal executions. For example, for a token passing system, we can define the legal executions to be those in which there is exactly one token in every state, and in which every process periodically receives a token.

What do we mean when we say that an automaton A stabilizes to the executions in set C. Intuitively, we mean that eventually all executions of A begin to "look like" an execution in set C. For example, suppose C is the set of legal executions of a token passing system. Then in the initial state of A there may be zero or more tokens. However, the definition requires that eventually there is some suffix of any execution of A in which there is exactly one token in any state.

Formally:

Definition 2.1 *Let C be a set of executions. We say that automaton A stabilizes to the executions in C if for every execution α of A there is some suffix of execution α that is in C.*

We can extend this definition in the natural way to define what it means for an automaton A to stabilize to the executions of another automaton B.

3 Dijkstra's Shared Memory Model

In Dijkstra's [Dij74] model, a network protocol is modeled using a graph of finite state machines. In a single move, a *single* node is allowed to read the state of its neighbors, compute, and then possibly change its state. In a real distributed system such atomic communication is impossible. Typically communication has to proceed through channels. While Dijkstra's original model is not very realistic, it is probably the simplest model of an asynchronous distributed system. This simple model provided an ideal vehicle for *introducing* [Dij74] the concept of stabilization without undue complexity. We will use Dijkstra's original model to *introduce* two important and general techniques for self-stabilization: *local checking and correction* and *counter flushing*.

3.1 Dijkstra's Shared Memory Model and IOA

How can we map Dijkstra's model into the IOA (I/O Automaton) model? Suppose each node in Dijkstra's model is a separate automaton. Then in the Input/Output automata model, it is not possible to model the simultaneous reading of the state of neighboring nodes. The solution we use is to dispense with modularity and model *the entire network as a single automaton.* All actions, such as reading the state of neighbors and computing, are *internal actions.* The asynchrony in the system, which Dijkstra modeled using a "demon", is naturally a part of the IOA model. Also, we will describe the correctness of Dijkstra's systems in terms of *executions* of the automaton.

Formally:

A *shared memory network automaton* $\mathcal{N}$ for graph $G = (E, V)$ is an automaton in which:

- The state of $\mathcal{N}$ is the cross-product of a set of node states, $S_u(\mathcal{N})$, one for each node $u \in V$. For any state s of $\mathcal{N}$, we use $s|u$ to denote s projected onto S_u. This is also read as the state of node u in global state s.
- All actions of $\mathcal{N}$ are internal actions and are partitioned into sets, $A_u(\mathcal{N})$, one for each node $u \in V$
- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Consider any state s' of $\mathcal{N}$ such that $s'|u = s|u$ and $s'|v = s|v$ for all neighbors v of u. Then there is

some transition $(s', \pi, \tilde{s}')$ of $\mathcal{N}$ such that $\tilde{s}'|v = \tilde{s}|v$ for u and all u's neighbors in G.

- Suppose $(s, \pi, \tilde{s})$ is a transition of $\mathcal{N}$ and π belongs to $A_u(\mathcal{N})$. Then $s|v = \tilde{s}|v$ for all $v \neq u$.

Informally, the third condition requires that the transitions of a node $u \in V$ only depend on the state of node u and the states of of the neighbors of u in G. The fourth condition requires that the effect of a transition assigned to node $u \in V$ can only be to change the state of u.

4 Dijkstra's Second Example: Local Checking and Correction

In this section, we will begin by reconsidering the second example in [Dij74]. The derivation here is based on joint work [AGV94] with Anish Arora and Mohamed Gouda at the University of Texas. This protocol is essentially a token passing protocol on a line of nodes with process indices ranging from 0 to $n-1$. Imagine that the line is drawn vertically so that process 0 is at the bottom of the line (and hence is called "bottom") and Process $n-1$ is at the top of the line (and called "top"). This is shown in Figure 1. The down neighbor of Process i is Process $i-1$ and the up neighbor is Process $i+1$. Process $n-1$ and Process 0 are not connected.

Dijkstra observed that it is impossible (without randomization) to solve mutual exclusion in a stabilizing fashion if all processes have identical code. To break symmetry, he made the code for the "top" and "bottom" processes different from the code for the others.

Dijkstra's second example is modeled by the automaton $D2$ shown in Figure 2. The automaton is described a guarded command language: the preconditions describe guards, and the effects describe the state changes that occur when the action is taken.

Each process i has a boolean variable up_i, and a bit x_i. Roughly, up_i is a pointer at node i that points in the direction of the token, and x_i is a bit that is used to implement token passing. Figure 1 shows a state of this protocol when it is working correctly. First, there can be at most two consecutive nodes whose up pointers differ in value and the

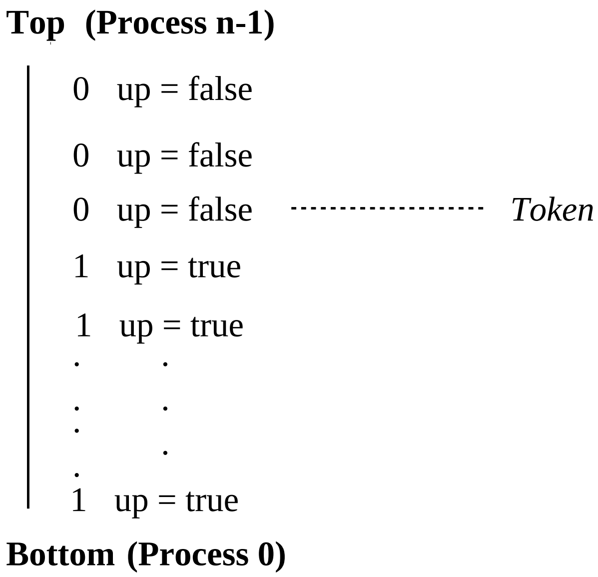

Figure 1: Dijktra's protocol for token passing on a line

token is at one of these two nodes. If the two bits at the two nodes are different (as in the figure) then the token is at the upper node; else the token is at the lower node.

For the present, assume that all processes start with $x_i = 0$. Also, initially assume that $up_i = \mathit{false}$ for all processes other than process 0. We will remove the need for such initialization below. We start by understanding the correct executions of this protocol when it has been correctly initialized.

A process i is said to have the token when any action at Process i is enabled. As usual the system is correct when there is at most one token in the system. Now, it is easy to see that in the initial state only MOVE_UP_0 is enabled. Once node 0 makes a move, then MOVE_UP_1 is enabled followed by MOVE_UP_2 and so on as the "token" travels up the line. Finally the token reaches node $n-1$, and we reach a state s in which $x_i = x_{i+1}$ for $i = 0 \ldots n-3$ and $x_{n-1} \neq x_{n-2}$. Also in state s, $up_i = \mathit{true}$ for $i = 0 \ldots n-2$ and $up_{n-1} = \mathit{false}$. Thus in state s, MOVE_DOWN_{n-1} is enabled and the token begins to move down the line by executing MOVE_DOWN_{n-2} followed by MOVE_DOWN_{n-3} and so on until we reach the initial state again. Then the cycle continues. Thus in correct executions, the "token" is passed up and down the line.

In the good states for Dijktra's second example, the line can be partitioned into two bands as shown in Figure 3. All bit and pointer values within a band are equal. (If a

The state of the system consists of a boolean variable
up_i and a bit x_i, one for every process in the line.
We will assume that $up_0 = \mathit{true}$ and $up_{n-1} = \mathit{false}$ by definition

In the initial state $x_i = 0$ for $i = 0 \ldots n-1$ and $up_i = \mathit{false}$ for $i = 1 \ldots n-1$

$\textsc{Move\_Up}_0$ (*action for the bottom process only to move token up*)
Precondition: $x_0 = x_1$ and $up_1 = \mathit{false}$
Effect: $x_0 :=\sim x_0$

$\textsc{Move\_Down}_{n-1}$ (*action for top process only to move token down*)
Precondition: $x_{n-2} \neq x_{n-1}$
Effects:
$x_{n-1} := x_{n-2}$;

$\textsc{Move\_Up}_i, 1 \leq i \leq n-2$ (*action for other processes to move token up*)
Precondition: $x_i \neq x_{i-1}$
Effects:
$x_i := x_{i-1}$;
$up_i := \mathit{true}$; (*point upwards in direction token was passed*)

$\textsc{Move\_Down}_i, 1 \leq i \leq n-2$ (*action for other processes to move token down*)
Precondition: $x_i = x_{i+1}$ and $up_i = \mathit{true}$ and $up_{i+1} = \mathit{false}$
Effect: $up_i := \mathit{false}$; (*point downwards in direction token was passed*)

Every continuously enabled action will eventually occur.

Figure 2: Automaton $D2$: a version of Dijkstra's second example with initial states. The protocol does token passing on a line using nodes with at most 4 states.

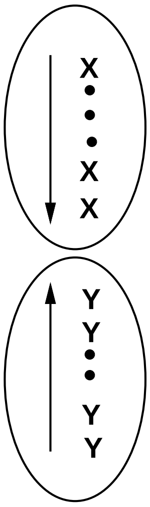

Figure 3: In the good states for Dijktra's second example, the line can be partitioned into 2 bands with the token at the boundary.

node within a band has $up_i = \mathit{false}$ we sketch it as a pointer that points downwards). All nodes within the upper band point downwards and all nodes within the lower band point upward. The token is at the boundary between the two bands. If the bit value X of the upper band is equal to the bit value Y of the lower band, the token is moving downwards; if the two bit values are unequal the token is moving downwards.

We describe these "good states" of $D2$ (that occur in correct executions) in terms of local predicates. In the shared memory model, a local predicate is any predicate that only refers to the state variables of a pair of neighbors. Intuitively, we see that if two neighboring nodes have the same pointer value, then their bits are equal; also if a node is pointing upwards, then so is its lower neighbor. Thus in a good state of $D2$, two properties are true for any Process i other than 0:

- If $up_{i-1} = up_i$ then $x_{i-1} = x_i$.
- If $up_i = \mathit{true}$ then $up_{i-1} = \mathit{true}$.

First, we prove that if these two local predicates hold for all $i = 1 \ldots n-1$, then there is exactly one action enabled. Intuitively, since $up_{n-1} = \mathit{false}$ and $up_0 = \mathit{true}$, we can start with process $n-1$ and go down the line until we find a pair of nodes i and $i-1$ such that $up_i = \mathit{false}$ and $up_{i-1} = \mathit{true}$. Consider the first such pair. Then the second predicate guarantees us that there is exactly one such pair. The first predicate then guarantees that all nodes $j < i-1$ have $x_j = x_{i-1}$ and all nodes $k > i$ have $x_k = x_i$. Thus only one action is enabled. If $x_i = x_{i-1}$ and $i-1 \neq 0$ then only MOVE_DOWN$_{i-1}$ is enabled. If $x_i = x_{i-1}$ and $i-1 = 0$ then only MOVE_UP$_0$ is enabled. If $x_i \neq x_{i-1}$ and $i \neq n-1$ then only MOVE_UP$_i$ is enabled. If $x_i \neq x_{i-1}$ and $i = n-1$ then only MOVE_DOWN$_{n-1}$ is enabled.

Similarly we can show that if exactly one action is enabled then all local predicates hold. Thus if the system is in a bad state, some pair of neighbors will be able to detect this fact. Hence we conclude that $D2$ is *locally checkable*. However, we would like to go even further and be able to correct the system to a good state by adding extra correction actions to each node. If we can add correction actions so that all link predicates will eventually become true, then we say that the $D2$ is also *locally correctable*. Adding local correction actions is tricky because the correction actions of nodes may "interfere" and result in a form of "thrashing".

However, a line is a special case of a tree with (say) 0 as the root; each node i other than 0, can consider $i-1$ to be its parent. The tree topology suggests a simple correction strategy. For each node $i \neq 0$, we can add a new action CORRECT_CHILD$_i$ (which is a separate class for each i). Basically, CORRECT_CHILD$_i$ checks whether the link predicate on the link between i and its parent is true. If not, i changes its state such that the predicate becomes true. Notice that CORRECT_CHILD$_i$ leaves the state of i's parent unchanged. Suppose j is the parent of i and k is the parent of j. Then CORRECT_CHILD$_i$ will leave the local predicate on link (j, k) true if it was true in the previous state.

Thus we have an important stability property: correcting a link does not affect the correctness of links above it in the tree. Using this it is easy to see that eventually all links will be in a good state and so the system is in a good state. We will ignore the details of this proof but the basic idea should be clear. Dijkstra's actual code does not use an explicit correction action. However, we feel that this way of understanding his example is clearer and illustrates a general method.

The general method of local checking and correction was first introduced by Awerbuch, Patt, and Varghese in [APV91b] and was used to solve somewhat more complex examples in a message passing model. After the introduction of this technique, I applied this method to the Arora-Gouda [AG90] reset protocol and the example from Dijkstra's paper described above; independently, Anish Arora and Mohamed Gouda followed the same path. A joint paper ([AGV94]) describes some of these results.

5 Dijkstra's first example: Counter Flushing

The counter flushing paradigm described in this section initially appeared in [Var93]. Many more applications of this technique are described in [Var94].

Dijkstra's first example is modeled by the automaton $D2$ shown in Figure 4. As in the previous example, the nodes (once again numbered from 0 to $n-1$) are arranged such that node 1 has node 0 and node 2 as its neighbors and so on. However, in this case we also assume that Process 0 and $n-1$ are neighbors. In other words, by making 0 and $n-1$ adjacent we have made the line into a ring. For process i, let us call Process $i-1$ (we assume that all arithmetic on indices and counters is mod n) the anticlockwise neighbor of i and $i+1$ the clockwise neighbor of i.

Each node has a counter $count_i$ in the range $0, \dots n$ that is incremented mod $n+1$. Once again the easiest way to understand this protocol is to understand what happens when it is properly initialized. Thus assume that initially Process 0 has its counter set to 1 while all other processes have their counter set to 0. Processes other than 0 are only allowed to "move" (see Figure 4) when their counter differs in value from that of their anticlockwise neighbor; in this case, the process is allowed to make a move by setting its counter to equal that of its anticlockwise neighbor. Thus initially, only Process 1 can make a move after which Process 1 has its counter equal to 1; next, only Process 2 can move, after which Process 2 sets its counter equal to 1; and so on, until the value 1 moves clockwise around the ring until all processes have their counter equal to 1.

Process 0 on the other hand cannot make a move until Process $n-1$ has the same counter value as Process 0. Thus until Process 1 sets its counter to 1, Process 0 cannot make a move. However, when this happens, Process 1 increments its counter mod $n+1$. Then the cycle repeats as now the value 2 begins to move across the ring (assuming $n > 2$) and so on. Thus after proper initialization, this system does perform a form of token passing on a ring; each node is again considered to have the token, when the system is in a state in which the node can take a move.

The good global states of the protocol can be sketched in Figure 5 Notice that the ring can be partitioned into 2 bands. All counter values within a band are equal and the

The state of the system consists of an integer variable
$count_i \in \{0, \dots n\}$, one for every process in the ring.
We assume that Process 0 and $n-1$ are neighbors

In the initial state $count_i = 0$ for $i = 1 \dots n-1$ and $count_1 = 1$

Move_0 (*action for Process 0 only *)
Precondition: $count_0 = count_{n-1}$ (*equal to anticlockwise neighbor?*)
Effect: $count_0 := (count_0 + 1) \bmod (n+1)$ (*increment counter*)

$\text{Move}_i, 1 \leq i \leq n-1$ (*action for other processes*)
Precondition: $count_i \neq count_{i-1}$ (*not equal to anticlockwise neighbor?*)
Effects:
$count_i := count_{i-1}$;(*set equal to anticlockwise neighbor*)

Every continuously enabled action will eventually occur.

Figure 4: Automaton $D1$: a version of Dijkstra's first example with initial states. The protocol does token passing on a ring using nodes with n states.

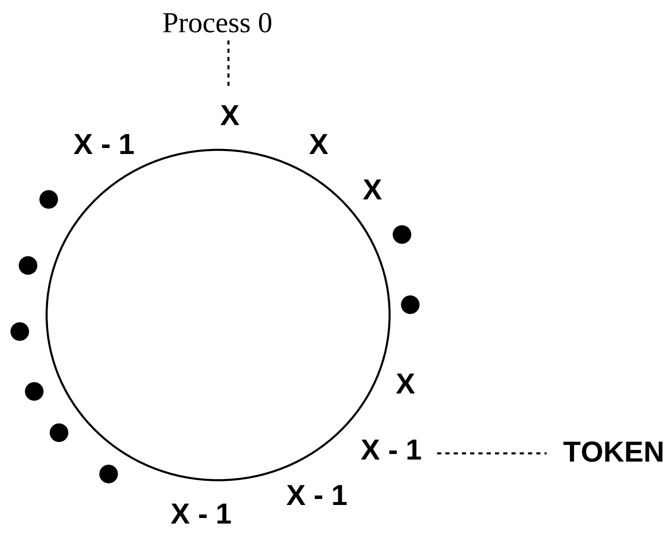

Figure 5: In the good states for Dijktra's first example, the ring can be partitioned into 2 bands with the token at the boundary.

band that includes the top node has a counter value one higher than the lower band. The token is at the boundary between the two bands; after that node makes a move, the top band becomes larger and the lower band becomes smaller.

It is easy to see that the system is in a good state iff the following local predicates are true.

- For $i = 1 \ldots n-1$, either $count_{i-1} = count_i$ or $count_{i-1} = count_i + 1$.
- Either $count_0 = count_{n-1}$ or $count_0 = count_{n-1} + 1$.

The system is locally checkable but it does not appear to be locally correctable. However, it does stabilize using a paradigm that we can call *counter flushing*. Even if the counter values are arbitrarily initialized (in the range $0, \ldots, n$) the system will eventually begin executing as some suffix of a properly initialized execution. We will prove this informally using three claims:

- **In any execution, Process 0 will eventually increment its counter.** Suppose not. Then since Process 0 is the only process that can "produce" new counter values, the number of distinct counter values cannot increase. If there are two or more distinct counter values, then moves by Processes other than 0 will reduce the number of distinct counter values to 1, after which Process 0 will increment its counter.

- **In any execution, Process 0 will eventually reach a "fresh" counter value that is not equal to the counter values of any other process.** To see this, note that that in the initial state there are at most n distinct counter values. Thus there is some counter value say m that is not present in the initial state. Since, process 0 keeps incrementing its counter, Process 0 will eventually reach m and in the interim no other process can set their counter value to m.

- **Any state in which Process 0 has a fresh counter value m is eventually followed by a state in which all processes have counter value m.** It is easy to see that the value m moves clockwise around the ring "flushing" any other counter values, while Process 0 remains at m. This is why I call this paradigm counter flushing.

The net effect is that any execution of $D1$ eventually reaches a good state in which it remains. The general counter flushing paradigm can be stated roughly as follows:

- A sender (in our case, Process 0) periodically sends a message to a set of responders; after all responders have received the message the sender sends the next message (this corresponds to one cycle around the ring in our case).

- To make the protocol stabilizing, the sender numbers each message with a counter. The size of the counter must be greater than the maximum number of distinct counter values that can be present in the network in any state. The sender increments its counter after the message has reached all responders.

- The protocol must ensure that the sender counter value will always increment and so eventually reach a "fresh" value not present in the network. A freshly numbered message m from the sender must guarantee that all old counter values will be "flushed" from the network before the sender sends the next message.

In Dijkstra's first example, the flushing property is guaranteed because the topology consists of a ring of unidirectional links. We have recently generalized the notion of counter flushing ([Var94]) to a range of other tasks and topologies.

6 Summary

Self-stabilization was introduced by Dijkstra in a seminal paper [Dij74]. Dijkstra's shared memory model is enchanting in its simplicity and makes an ideal vehicle to describe non-trivial examples of self-stabilization. We showed how to simulate Dijkstra's model as one big IOA with a few restrictions. We also used Dijkstra's first two examples to introduce two powerful methods for self-stabilization: counter flushing and local checking and correction.

Local checking has been used to state a variety of general theorems that are contained in my thesis [Var93]. For instance, one theorem generalizes the example given above (Dijkstra's second protocol) and shows that *any* locally checkable protocol that works on a tree topology can be made stabilizing. The thesis also defines the notion of *local correctability* — i.e., the class of protocols that can be corrected to good global states by local correction actions. Using this notion, a local correction theorem is stated which shows that any locally correctable protocol can be stabilized in time proportional to the height of a partial order that underlies the definition of local correctability. The second theorem applies to arbitrary graphs and to a general message-passing model. We have used these local checking theorems to provide solutions for mutual exclusion, the end-to-end problem, stabilizing synchronizers, and spanning tree protocols. More detailed references can be be found in [APV91b], [Var93] and [AV91].

Counter flushing has also been used as a design technique in several different contexts and applications. Other applications [Var94] include data link protocols, deadlock detection, network resets, and virtual circuit establishment protocols.

It is remarkable that the seed ideas behind these two powerful techniques are contained in Dijkstra's original paper.

References

[AG90] Anish Arora and Mohamed G. Gouda. Distributed reset. In *Proc. 10th Conf. on Foundations of Software Technology and Theoretical Computer Science*, pages 316–331. Spinger-Verlag (LNCS 472), 1990.

[AGV94] Anish Arora, Mohamed G. Gouda, and George Varghese. Distributed constraint satisfaction. In *Proc. 14th Conf. on Distributed Computing Systems (DCS)*, Poznan, Poland, 1994.

[APV91b] Baruch Awerbuch, Boaz Patt-Shamir, and George Varghese. Self-stabilization by local checking and correction. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[AV91] Baruch Awerbuch and George Varghese. Distributed program checking: a paradigm for building self-stabilizing distributed protocols. In *Proc. 32nd IEEE Symp. on Foundations of Computer Science*, October 1991.

[Dij74] Edsger W. Dijkstra. Self stabilization in spite of distributed control. *Comm. of the ACM*, 17:643–644, 1974.

[KP90] Shmuel Katz and Kenneth Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*,

[LT89] Nancy A. Lynch and Mark R. Tuttle. An introduction to input/output automata. *CWI Quarterly*, 2(3):219–246, 1989.

[Var93] George Varghese. Self-stabilization by local checking and correction. Ph.D. thesis, Lab for Computer Science, MIT, MIT/LCS/TR-583, October 1993.

[Var94] George Varghese. Self-stabilization by Counter Flushing. To appear in the *Proceedings of the 13th PODC*, Los Angeles, California, August 1992. October 1993.