SEVER INSTITUTE OF TECHNOLOGY

MASTER OF SCIENCE DEGREE

THESIS ACCEPTANCE

(To be the first page of each copy of the thesis)

DATE: 12 July 1996

STUDENT'S NAME: Adam M. Costello

This student's thesis, entitled Self-Stabilization by Counter Flushing and Window Washing has been examined by the undersigned committee of three faculty members and has received full approval for acceptance in partial fulfillment of the requirements for the degree Master of Science.

APPROVAL: ______________________ Chairman

____

____

Short Title: Counter Flushing and Window Washing Costello, M.Sc. 1996

WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

SELF-STABILIZATION BY COUNTER FLUSHING AND WINDOW WASHING

by

Adam M. Costello

Prepared under the direction of Professor George Varghese

A thesis presented to the Sever Institute of
Washington University in partial fulfillment
of the requirements for the degree of

Master of Science

August, 1996

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

ABSTRACT

SELF-STABILIZATION BY COUNTER FLUSHING AND WINDOW WASHING

by Adam M. Costello

ADVISOR: Professor George Varghese

August, 1996
Saint Louis, Missouri

A self-stabilizing protocol is one that can automatically resume correct operation after arbitrary corruption of protocol state variables and the injection of arbitrary messages. We demonstrate the advantages and feasibility of self-stabilization by modifying the Fiber Distributed Data Interface Media Access Control (FDDI MAC), a widely used token-ring protocol, to use *counter flushing*, an existing technique for achieving self-stabilization. Our simple modifications not only make the protocol more robust (for example, our protocol recovers from multiple tokens in less than 5.8 ms, while the original MAC might never recover), but also allow it to recover from lost tokens more quickly (0.001–0.36 ms versus 2.5–4.1 ms).

Token passing is equivalent to non-pipelined broadcast, but we have fashioned a generalization of counter flushing called *window washing* which applies to *pipelined*

broadcast. The results are simple, efficient, self-stabilizing sliding window protocols which can be used for data links, reliable multicast, and flow control.

Contents

List of Tables

List of Figures

Acknowledgments

Many thanks to my advisor, George Varghese, for encouraging me to apply to the graduate program, for showing me the ropes, for suggesting interesting problems, and for preparing me for doctoral research.

Innumerable thanks to my parents, Martin and Wendy Costello, for their love and support, and for being admirable examples.

Adam M. Costello

Washington University in Saint Louis

August 1996

Chapter 1

Introduction

1.1 Overview

A protocol is *self-stabilizing* if, when started from an arbitrary global state, it exhibits "correct" behavior in bounded time. Typical protocols cope with a specified set of failure modes such as packet loss and link failures. A self-stabilizing protocol copes with a set of failures that subsumes most previous categories, and is robust against transient errors, including memory corruption and the injection of corrupt packets. Transient errors do occur in real networks and cause systems to fail unpredictably [16, 14]. Thus stabilizing protocols are more *robust* than traditional protocols. Furthermore, they are *simpler*, because they use uniform mechanisms to deal with different failures, and because they do not need special mechanisms for initialization.

Varghese had earlier [22] developed a technique called *counter flushing* which provided (in theory) a simple self-stabilizing token-passing protocol. We have adapted this technique to work with the complexities and performance concerns of a real-world token-passing protocol, the FDDI Media Access Control (MAC) [2]. The theoretical protocol did not take into account the distinction between the ring latency and the token rotation time, nor the requirement to minimize buffering at each node to less than the size of a token.

Counter flushing also provides a simple self-stabilizing protocol that allows a single leader to broadcast one message at a time to the receivers. Such *stop-and-wait* protocols are inefficient in networks in which the propagation delay is large compared to the transmission delay. For the simplest case of one sender and one receiver, the sliding window protocol [20] allows the sender to pipeline a *window* of packets, improving throughput. We have generalized counter flushing to work with the sliding window protocol, calling the new technique *window washing*. We have also generalized the sliding window protocol to the case of one sender and multiple receivers, and applied window washing to make the protocol self-stabilizing. One-to-many sliding window protocols would be useful for reliable multicast, for example in ATM or IP networks. To the best of our knowledge, even the non-stabilizing version of our one-to-many sliding window protocol is new.

The theoretical notions underlying window washing (WW) nicely generalize the notions underlying counter flushing (CF). CF requires a space of sequence numbers that is larger than the number of messages that can be stored in the system. WW requires a space larger by a multiplicative factor equal to the degree of pipelining (the window size). Stabilization in both CF and WW is achieved by having initial garbage values flushed out of the system, replaced by values generated by a *leader*, during one round-trip delay, and then having the leader get "stuck" until the rest of the participants "catch up" during the second round-trip delay.

Sliding windows are not only used to provide reliability for data transmission. With minor modifications, they can also be used for flow control, and window washing works here as well to achieve self-stabilization. We show how this leads to a self-stabilizing flow control protocol almost (but not quite) isomorphic to the non-stabilizing Credit Update Protocol (CUP) [9], which has been implemented in Bell-Northern Research ATM switches.

1.2 Related Work

Self-stabilizing protocols were introduced by Dijkstra [4, 17]. There have been few general techniques for self-stabilization. Katz and Perry [8] showed how to compile an arbitrary asynchronous protocol into a stabilizing equivalent. The basic idea is to add a leader node which periodically takes a snapshot of the global state, checks for inconsistencies, and initiates a reset if any are found. The transformation is expensive, so more efficient and less general techniques are useful.

For some protocols, global inconsistencies can be detected using local checking. Techniques that transform any *locally checkable* protocol into a stabilizing equivalent are given in [1, 3, 21]. However, our work on counter flushing and window washing applies to protocols that are not locally checkable.

Gouda and Multari [6] describe a two-node sliding window protocol using *unbounded* integers. In any real implementation, there will be a limited number of bits available to represent the integers. Hence, in order to be useful, protocols must work with bounded integers.

Spinelli [18, 19] describes a pair of two-node self-stabilizing sliding window protocols with bounded counters. The sender keeps an extra internal one-bit register, which it periodically inverts. The sender includes a copy of this bit in the data packets, the receiver copies it into the corresponding acks, and the sender checks for the proper bit value when processing acks. If the period of the bit inversion at the sender exceeds the *worst-case* round-trip delay, the protocol stabilizes in two round-trip delays[1], one of which is a worst-case round-trip delay. Our protocols stabilize faster, taking at most two *actual* round-trip delays to stabilize. In some real links the worst-case round-trip delay exceeds the typical case by an order of magnitude. The cost of this faster stabilization is a few bits in the packet headers—we add $\lg c_{max}$ bits (c_{max} is roughly the number of packets that fit in the links) whereas Spinelli adds only one. Our protocol stabilizes using a completely different mechanism from Spinelli's, one that can be applied in other contexts. For instance, we apply our mechanisms to the general case of *one-to-many* sliding window protocols, and to flow control protocols.

Work on distributed algorithms has explored the Byzantine fault model [10], in which a protocol is required to operate correctly even while a fraction of the nodes are arbitrarily misbehaving. Self-stabilization, on the other hand, allows *all* nodes and links to misbehave, but also allows the protocol to operate incorrectly until some finite time after the faults cease. The two models are therefore orthognal, and could even be combined, but this thesis focuses on self-stabilization.

A few protocols used in real networks are self-stabilizing. The Update Process in OSI Network Layer Protocol 10589 broadcasts routing updates to all nodes in a

[1] In the go-back-n version of Spinelli's protocol, $RESET$s are not possible; in the selective repeat version (a feature we do not explore), $RESET$s add another worst-case round-trip delay plus an actual round-trip delay to the stabilization time.

network. Its main ideas are described in [14] and [15]. The IEEE 802.1 Spanning Tree Protocol, which computes a shortest-path tree among data link layer bridges, is described in [13]. Both protocols achieve stabilization by sending state information to neighbors periodically (even when the state has not changed), and by timing out old information. Both protocols have long stabilization times, because the timers must be set proportional to worst-case network delays rather than actual network delays.

The SNR Transport Protocol [12] also achieves self-stabilization by periodically sending state information. It is simple and effective, but limited to point-to-point connections. Our window-washing method will work with one-to-many connections.

1.3 Organization

The rest of the thesis is organized as follows. Chapter 2 first defines the network model used for both counter flushing and window washing, then reviews counter flushing, including its application to token passing. Chapter 3 presents the modified FDDI MAC protocol using a variation of the counter flushing technique, and proves its correctness. Chapter 4 develops window washing in the context of two-node reliable data transmission, proves its correctness, and extends its application to reliable multicast and to flow control. Chapter 5 states our conclusions.

Chapter 2

Counter Flushing

This chapter reviews counter flushing [4, 22]. The abstract ideas are:

- Nodes periodically inform their neighbors of "where" in the execution they are.
- There is a unique leader node who is always "right" by definition.
- Whenever a discrepancy is discovered, the leader ignores it, but a non-leader will re-sync by altering its own state.

The specifics of the technique are organized as follows. Section 2.1 clarifies our network model, which is also used for window washing (in Chapter 4). Section 2.2 illustrates counter flushing in the context of a token ring. Section 2.3 explains how the ring stabilizes.

2.1 Network Model

We restrict ourselves to message-passing protocols for networks. The network topology is modeled by a directed graph $G = \langle V, E \rangle$. Let $n = |V|$ denote the number of network nodes and D the network diameter. We assume that there is a distinguished leader node. (There are many stabilizing protocols to construct a leader; e.g., [13] calculates a leader in $O(D)$ time.) We model the nodes and links of the network using Input/Output Automata (IOA) [11].

Nodes communicate with each other by sending and receiving *packets* to and from links. Nodes and links are modeled by state machines. For every link $\langle i, j \rangle \in E$, there is an output action $\mathsf{Send}_{i,j}(p)$ to send a packet p from node i toward node j, and an input action $\mathsf{Receive}_{i,j}(p)$ to receive a packet[1] at node j from node i. Similarly, the link itself has an input action $\mathsf{Send}_{i,j}(p)$ to receive packets from node i, and output action $\mathsf{Receive}_{i,j}(p)$ to deliver packets to node j. We will generally give the actions more intuitive aliases, and split up the packet p into constituent fields; for example, $\mathsf{Send}_{0,1}(p)$ and $\mathsf{Send}_{1,0}(p)$ might be referred to as $\mathsf{SendData}(seq, m)$ and $\mathsf{SendAck}(ack)$.

Node automata can be arbitrary except that they must have *finite* state sets and have the appropriate interface actions to send and receive packets. We assume that each link is a FIFO queue with *initially bounded storage*; that is, there may be no more than L_{max} packets on a link in the initial state. (This is a reasonable assumption because all real links are bounded.) To model link errors, a $\mathsf{Send}_{i,j}(p)$ action may result in no change in the queue. However, we assume that a sequence of $\mathsf{Send}_{i,j}(p)$ actions will eventually result in some action succeeding and storing the last packet sent at the tail of the link queue. A $\mathsf{Receive}_{i,j}(p)$ action is enabled whenever p is at the head of the link queue; it removes p from the queue.

2.2 Token Passing

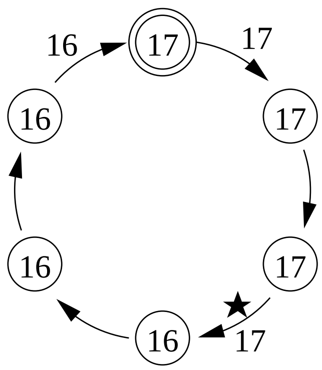


Figure 2.1: A valid state of a token ring using counter flushing.

[1]The convention for action subscripts is that the first subscript always represents the sending node, and the second the receiving node.

The details of counter flushing are best illustrated using a token ring. Number the nodes sequentially clockwise, starting with 0 at the leader. As in a normal ring, a token is sent (clockwise) around the ring such that at most one node has the token at a time. For fault-tolerance, let each token packet carry a counter value c indicating how many times the token has circumnavigated the ring, and let every node i store the counter value c_i of the last token that arrived at the node. To guard against lost tokens, each node i periodically retransmits a token packet containing c_i. Nodes must use their counters to weed out duplicate tokens. In a valid state, there are only two values in the ring: one arc, starting with the leader and continuing part-way around the ring, contains only the value c_0, while the rest of the ring contains only the value $c_0 - 1$. The token is at the end of the first arc. For example, Figure 2.1 shows a valid state with the token in transit from node 2 to node 3.

All arithmetic is modulo M where $M > c_{max}$.

$\mathsf{Send}_{ij}(c)$ (* Node emits counter value. *)
  Preconditions:
    $c = c_i$

$\mathsf{Receive}_{ij}(c)$ (* Non-leader j absorbs counter value. *)
  Effects:
    if $c \neq c_j$ then I have the token
    $c_j \leftarrow c$

$\mathsf{Receive}_{i0}(c)$ (* Leader absorbs counter value. *)
  Effects:
    if $c = c_0$ then
      I have the token
      increment c_0

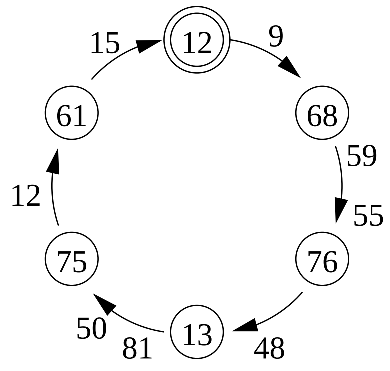
Figure 2.2: Token passing code with counter flushing.

When a non-leader node i receives a token packet from upstream, it compares the counter value in the packet (c) with its own counter value (c_i). If they are *different*, the node assumes it has received the token and sets c_i equal to c, but if they are *equal*, the node ignores the token packet. The leader, however, follows a different rule: if the counter value c in the packet is *equal* to the leader's counter value c_0, the leader assumes it has received the token and increments its counter value modulo M, but if c is *different* from c_0, the leader ignores the token packet. For example, from the state shown in Figure 2.1, the value 17 will sweep around the ring as the token moves clockwise. When the token returns to the leader, the leader increments to 18 and begins a new cycle. The protocol code is given in Figure 2.2.

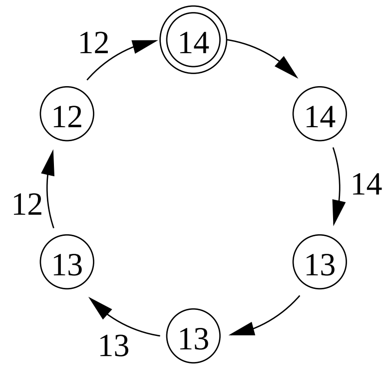
2.3 Stabilization

In the initial state, the counter values at nodes and in tokens stored on links may be arbitrary. Let c_{max} be the maximum number of counters that can be stored in the network in the initial state. (In our model, $c_{max} = |E|L_{max} + |V|$.) If the modulus M is greater than c_{max}, then the protocol stabilizes within two round-trip delays, where a round-trip delay is the time for a causal chain of packets to propagate all the way around the ring. Once the first leader-produced value has made its way all the way around the ring and is about to arrive back at the leader, the initial garbage values have all been flushed out, and only values produced by the leader are left in the ring. Since the leader could have incremented at most once for each distinct value initially in the ring, and $M > c_{max}$, the leader's counter value cannot appear anywhere except in a single arc starting at the leader. Therefore, the leader will not increment again until its current value has filled the ring, which takes at most one more round-trip delay, at which point the ring is in a valid state. Figure 2.3 shows an example stabilization scenario.

In the initial state, the ring is full of garbage:



After one round-trip delay, all garbage values have been flushed out:



The leader remains stuck until its own value has filled the ring (at most one more round-trip delay):

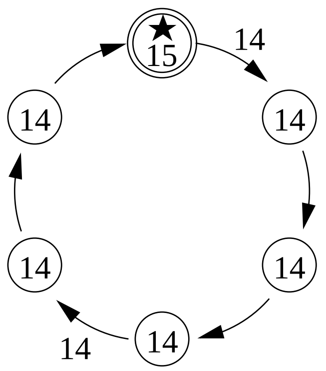


Figure 2.3: Stabilization of the token ring using counter flushing.

Chapter 3

Modified FDDI MAC

The self-stabilizing token-passing protocol described in Chapter 2 is remarkably simple, but when one tries to apply the technique to a real-world token ring, new concerns require more careful treatment. Section 3.1 reviews FDDI. Section 3.2 identifies the new concerns. Section 3.3 presents modifications to the FDDI MAC specification to incorporate counter flushing while satisfying the new requirements. Section 3.4 proves that the new protocol stabilizes. Sections 3.5, 3.6, and 3.7 analyze the costs, benefits, and simulation results, respectively.

3.1 Overview of FDDI

3.1.1 Normal Operation

Fiber Distributed Data Interface (FDDI) is a local area network standard using a token ring architecture. Up to 1000 stations are connected in a ring by up to 200 km of glass fiber. Data is transmitted at a rate of 25 million *Symbols* per second. There are sixteen data symbols (0–9, A–F), each representing a group of 4 bits, and 8 control symbols (H, I, J, K, Q, R, S, T). The data rate is therefore 100 Mbps.

Data bytes are grouped into *Frames* with headers and trailers, which contain delimiters, message type information, source and destination addresses, the payload, a checksum, and status flags. Only the station holding the token may send Frames—the other stations repeat the Frames, and any station (particularly the one matching the destination address) may copy the Frame and deliver the data up to a higher layer. When the token-holder is done transmitting, it relinquishes the token by sending a short *Token* message containing only the delimiters and message type. Another station may *capture* the token by deliberately failing to repeat the entire Token message.

FDDI provides latency bounds using *timed token access.* There is a *Target Token Rotation Time* (TTRT) agreed upon by all stations in the ring, and the stations cooperate to insure that the token circulates at least once every TTRT. There are two classes of Frames—synchronous and asynchronous. Each station may have a *synchronous allocation*, which is a percentage of the TTRT[1] Whenever a token arrives, a station may capture it and send synchronous Frames for its allocated time. After that, it may send asynchronous Frames, but only as long as the time since the previous token arrival is less than TTRT. If TTRT was already exceeded after the synchronous Frames were sent, no asynchronous Frames may be sent this time around. In this way, latency-sensitive (synchronous) traffic takes priority over other traffic. The actual token rotation time is provably TTRT on average, and $2 \times$ TTRT at most.

To prevent Frames from circulating around the ring multiple times, causing duplicate deliveries at the receiving station, Frames are *stripped* by their senders. Every station, while repeating a Frame, inspects the source address. If it sees its own address, the station stops repeating, and sends Idle Symbols (I) instead. The result is a *fragment* (partial message) which circulates harmlessly around the ring until it encounters a transmitting station, which does not repeat it.

3.1.2 Fault Recovery

At the physical layer, FDDI can used star-shaped rings, in which each link between adjacent stations makes a trip through a central device called a *concentrator*, which can short-circuit a station when it crashes, or when its links break.

At the MAC layer, each station has a Valid Transmission Timer (TVX) and a Late Counter (Late_Ct), which detect faults. TVX is reset to -2.5 ms each time a valid Frame is received. If it ever counts up to 0, the station will reset the ring (see below).

[1]Higher layers insure that the sum of the synchronous allocations do not exceed 100% of the TTRT.

Late_Ct counts multiples of TTRT since the token last arrived at the station. If it reaches 2, something must be wrong, so the station will reset the ring.

A station resets the ring by continually issuing *Claim Frames* containing the station's desired TTRT. Any station, upon receiving a Claim Frame from another station, repeats it, or, if its own desired TTRT is smaller, begins issuing its own Claim Frames (source addresses are used as tie-breakers). Eventually, the station with the smallest bid receives its own Claim Frame, knows that it has won the bidding, and issues a Token. This reset mechanism is called the *Claim Token process.*

If a station sends Claim Frames for a long time without receiving a lower bid or its own Claim Frame, it initiates the *Beacon process* to locate the presumed break in the ring. It continually sends *Beacon Frames.* Any station, upon receiving a Beacon Frame from another station, repeats it. Therefore, only the station immediately downstream of the break will continue to send Beacon Frames, a fact which can be useful in locating the break. If a station sending Beacon Frames receives its own Beacon Frame, it knows that the physical ring has been restored, and resets the ring using the Claim Token process.

A lost token will always be recovered. Usually, its absence will be detected by TVX, because if no station has the token, no station may create any Frames, and existing Frames should get stripped. If somehow a Frame exists with a source address matching none of the stations on the ring (perhaps caused by line noise) and the token is lost (perhaps also caused by line noise), then the Frame will circulate repeatedly, preventing any station's TVX from expiring. The Late Counter, however, will catch the fault, and cause the ring to be reset.

The problem of multiple tokens, on the other hand, might never be corrected. This situation might arise, for example, if a Token fragment is corrupted by line noise back into a Token. A Token fragment is a Token with its last three symbols missing, the result of a Token being captured by a station. Once there are two tokens in the ring, both will persist until a Token message encounters a token-holder, but there is no bound on how long that could take. Both tokens could propagate around the ring at equal average rates. Meanwhile, any data Frame sent by one token-holder to a station on the far side of the other token-holder will not reach its destination, because the second token-holder will not repeat it. Because Tokens and Frames are circulating, TVX and the Late Counter will not detect a problem.

3.2 Concerns of Real Token Rings

The counter flushing protocol described in Chapter 2 requires all nodes, even those lacking a token, to send their counter values periodically. In a real ring, this raises the problem of what to do if a frame arrives while the counter value is being sent. Enqueuing the incoming frame would add latency at each station in the ring, which is unacceptable. FDDI goes out of its way to make sure that there is no need for more than a single symbol's worth of latency at each node, because a ring may contain up to 1000 stations. Another option, interrupting the outgoing counter, would make it difficult to insure that the counter ever gets sent—perhaps every attempt will be interrupted.

The solution we use is to include a counter value in every frame. Each non-leader sends its counter value once when it gives up the token, but does not periodically resend it. Instead, only the leader resends its counter value, and non-leaders let those values pass through. In order for a node to capture a token, it must prevent the counter value from reaching nodes downstream from itself. It could do this either by replacing the passing counter value with its old counter value, or by clearing a special bit in the message.

This choice highlights a second new concern: the distinction between the ring latency and the token rotation time. In the theoretical protocol, they were one and the same, but in a real ring, the token rotation time can be *much* greater than the ring latency. Therefore, we must allow counter values to pass through a station holding a token, so we add a special bit to the messages which the token-holder can clear to prevent the message from carrying a token to the next station. When the bit is set, the message might or might not carry a token, but when the bit is unset, it certainly does not.

3.3 Specification Changes

3.3.1 Leader Selection

The theoretical counter flushing protocol assumes the existence of a unique leader node. In a real token ring, we must elect the leader, which means we need a self-stabilizing leader election algorithm. Such algorithms exist, and we believe we could adapt them to the FDDI MAC; however, we have not yet done this.

We have set aside the leader election problem as an orthogonal concern, and for the time being we use the FDDI MAC's existing Claim Token process as a non-stabilizing leader election algorithm. In the modified protocol, the winner of that process not only creates the initial Token, but also becomes the Leader. Because the Claim Token process is non-stabilizing, the modified protocol does not necessarily recover from conditions that violate the model's assumption of a unique Leader node. For example, a crashed Leader can leave the ring with zero Leaders and multiple tokens; or memory corruption or line noise could cause more than one station to believe it had won the Claim Token process, resulting in a ring with multiple Leaders. Replacing the Claim Token process with a self-stabilizing leader election algorithm would solve this shortcoming.

3.3.2 Token Class Simplification

The next modification is the removal of support for the *restricted* class of tokens. According to [7], restricted mode "is not known to be used by anyone". Supporting it probably would have been possible, but would have presented an uninteresting complication.

3.3.3 Message Format

Next we modify the message format, as shown in Figure 3.1. In an FDDI ring, every valid message is either a Token or a Frame, and is composed of Symbols. Both Tokens and Frames begin with a Start Delimiter (2 Symbols), followed by a Frame Control sequence (2 Symbols), which identifies the message type (Token, data Frame,

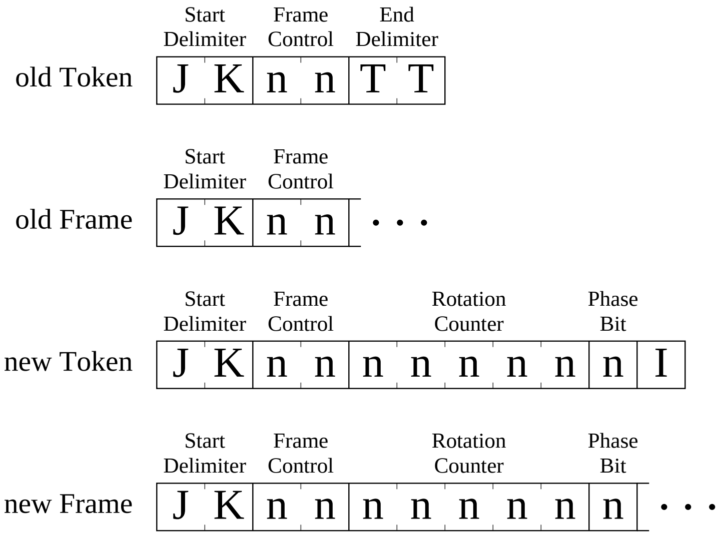


Figure 3.1: Comparison of old and new Token and Frame formats.

Claim Frame, etc.). A Token has an End Delimiter immediately following the Frame Control, while a Frame has several fields which do not concern us. In both Tokens and Frames, immediately following the Frame Control, we add a new Rotation Counter field (5 Symbols), which encodes a 12-bit counter in 20 bits for error detection. Then comes the new Phase Bit field (1 Symbol), which encodes the aforementioned "special bit" mentioned above in 4 bits for error detection. We replace the End Delimiter[2] in Tokens by an Idle Symbol, so that a Token looks like a fragment. Tokens and Frames will be treated identically by receivers. The Rotation Counter and Phase Bit together take the place of the single "counter value" of the theoretical technique.

Now a station can aquire the token upon receipt of a Frame as well as a Token. A Token does not necessarily carry a token—it just carries a Rotation Counter and Phase Bit which may or may not represent a token. (Watch carefully the capitalization of "Token" and "token"!) A station can also acquire the token upon receipt of a fragment. A message with an unset Phase Bit never carries a token.

[2]The Token End Delimiter was formerly used to confirm the validity of a Token, but in the modified protocol, that determination must be made as soon as the Phase Bit is received, before it is repeated.

3.3.4 Station Behavior

Each station contains an internal register with a rotation counter and a phase bit[3]. Any station, when creating a Token, copies its internal rotation counter and phase bit into the Token. (Non-Leaders create Tokens under exactly the same circumstances as in the old FDDI MAC spec—only when they are finished transmitting. The Leader additionally creates Tokens whenever it is idle.) Any station, when creating a Frame, copies its internal rotation counter into the Frame, and clears the Frame's Phase Bit regardless of its internal phase bit, so that it cannot give away the token while it is transmitting.

Non-Leader Behavior: Frame repetition is handled differently by the Leader and the non-Leaders. Non-Leaders copy all incoming Rotation Counter and Phase Bit fields into their registers (if no errors are detected in either). If this changes the register, and if the phase bit is now set, then the message carries a token, which the station may optionally capture by clearing the outgoing Phase Bit.

Leader Behavior: The Leader, on the other hand, copies in the other direction—from the internal register into the Rotation Counter and Phase Bit of the message. It also compares the incoming Rotation Counter and Phase Bit with its internal values. If they are equal, the message carries a token, and the Leader increments its internal rotation counter. The Leader cannot help but capture the token, but can release it just as effortlessly. The Leader's internal phase bit must *always* be set[4], even during stabilization, so it actually uses the constant 1 rather than a register.

The Leader transmits Tokens rather than Idle Symbols when it has nothing else to send or repeat (hence the effortless release). This requires it to transmit into a queue rather than directly onto the medium. The queue must be large enough to hold one Token (27 Symbols, including the required 16 Idle Symbols of Preamble[5]). Symbols are dequeued onto the medium. Whenever the queue becomes empty, it is loaded

[3]Lower case is used here because the internal register does not necessarily use the same encoding as the Rotation Counter and Phase Bit fields of Frames and Tokens.

[4]Conceptually, a set internal phase bit means "I have seen the token this round". Because each new round begins when the token returns to the Leader, the Leader's phase bit is always set.

[5]It *may* be necessary for the leader to follow the token with 15 extra Idle Symbols, to provide a full Preamble for the next repeated Frame, in which case the buffer must hold 42 Symbols.

with a Token. Thus, the Leader keeps the ring filled with messages. The largest gaps occur when Frames are stripped by non-Leaders, but the size of those gaps cannot exceed the maximum Frame size (9000 Symbols).

The subset of station behavior pertinent to this section is shown in pseudocode in Figure 3.2.

3.3.5 Ring Operation

All remaining aspects of the original specification remain unchanged, particularly rules regarding when tokens may be captured, how long they may be held, priorities, frame stripping, resets, statistics counters, and Beacons.

In normal operation, there will be no more than two rotation counter values in the ring, one stretching from the Leader to some point, and the previous one in the remainder of the ring. Also, phase bits will be set between the leader and the token holder (who has the same rotation counter as the leader), unset between the token holder and the boundary between the new and old rotation counter values, and set in the remainder of the ring. The token holder will send Frames with unset Phase Bits for a while, then send a Token with a set Phase Bit. This Phase Bit will be copied into successive stations until it arrives at one willing to capture the token. This process is repeated until the set Phase Bit makes its way back to the leader, which increments its rotation counter, starting the whole cycle over again. The new rotation counter gets copied into all the stations in about one ring latency, even though it typically takes the the token much longer to traverse the ring. Therefore the third arc, containing old rotation counter values and set phase bits, is usually empty. See Figure 3.3.

In correct operation, a station holding a token should never receive a token, nor should its phase bit be 0. Therefore we require a token holder to issue a Token immediately if either of these conditions is violated. While not strictly necessary, this policy speeds up the stabilization process.

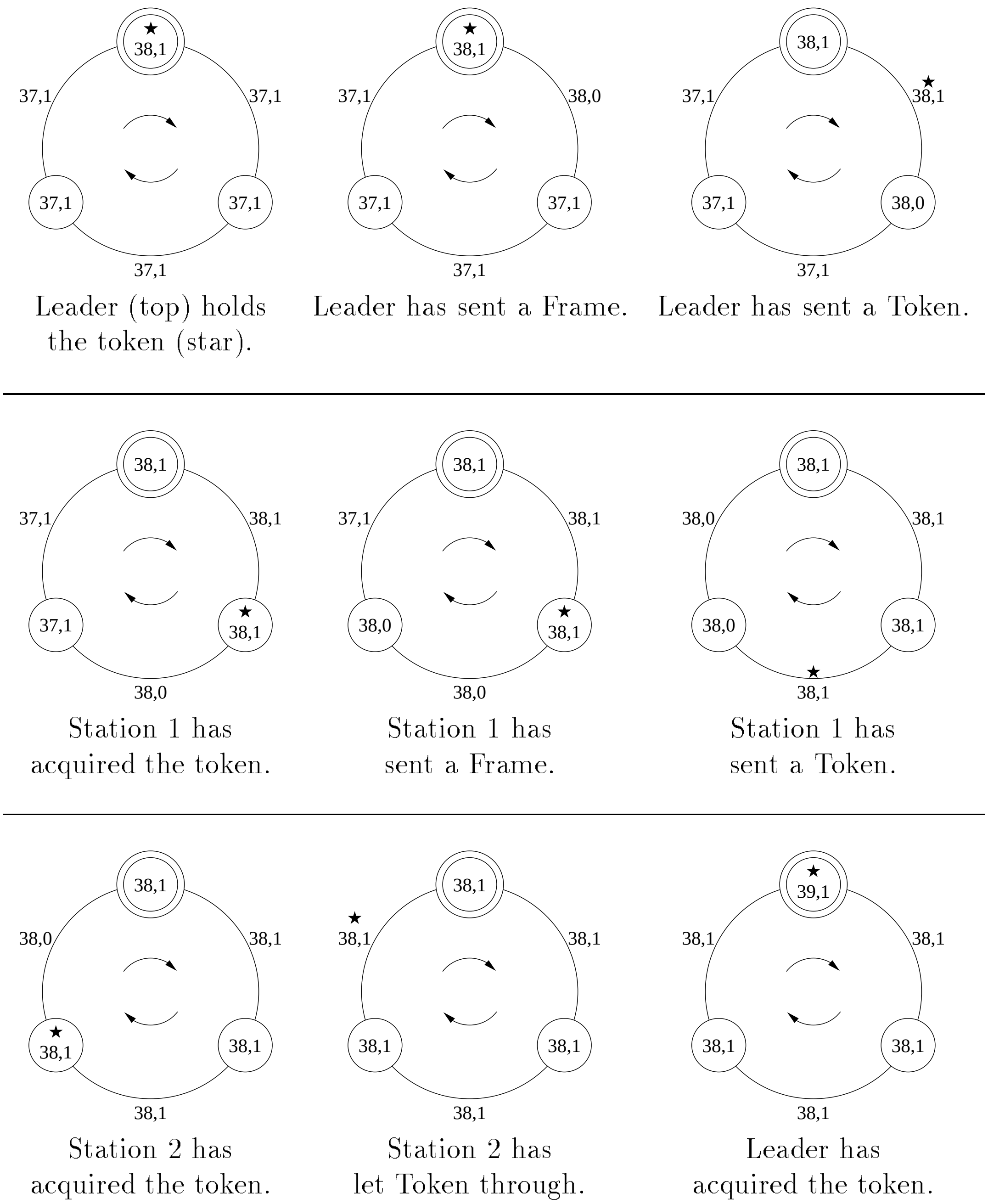
Receiver

```
while true do
    wait for start of message
    send start-of-message signal to transmitter
    if Rotation Counter and Phase Bit are error-free then
        if I am Leader then
            if Rotation Counter and Phase Bit equal internal register then
                increment internal rotation counter
                send token-received signal to transmitter
        else
            copy them into internal register
                if register changes and Phase Bit is set then
                    send token-received signal to transmitter
    wait for end of message or Source Address equal to my address
    send end-of-message signal to transmitter
```

Transmitter

```
while true do
    repeat
        if I am Leader then
            whenever queue is empty, send Token carrying
                internal Rotation Counter and a set Phase Bit
        else
            send Idle symbol
    until start-of-message signal
    repeat message until any signal (if Leader, overwrite
        Rotation Counter and Phase Bit from internal register)
    if token-received signal and token is usable then
        if I am Leader then send a set Phase Bit
        else send an unset Phase Bit
        while I am allowed to hold token
            send Frames with unset Phase bits, copying
                Rotation Counter from internal register
            break on token-received signal or if internal phase bit is unset
        send Token, copying Rotation Counter and Phase Bit from internal register
```

Figure 3.2: Pseudocode for station behavior. Each station contains a receiver and a transmitter running in parallel. Writes/reads to/from the internal register must be atomic.



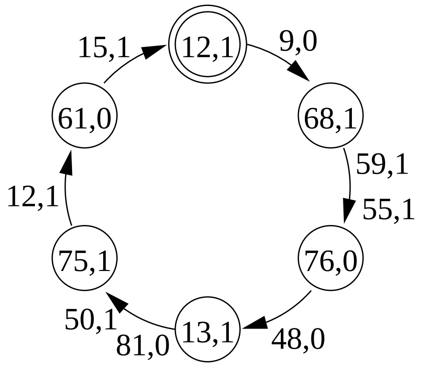
Leader (top) holds the token (star). Leader has sent a Frame. Leader has sent a Token.

Station 1 has acquired the token. Station 1 has sent a Frame. Station 1 has sent a Token.

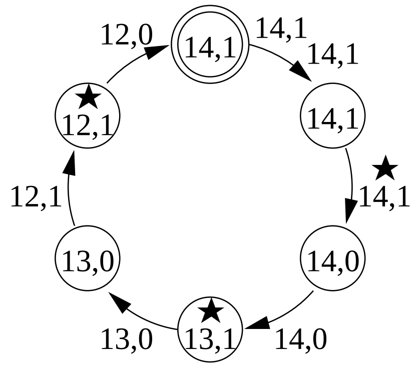
Station 2 has acquired the token. Station 2 has let Token through. Leader has acquired the token.

Figure 3.3: Example operation of the modified FDDI ring. Notice that the rotation counter propagates quickly around the ring, while the token goes more slowly, riding the boundary between the set and unset phase bits.

In the initial state, the ring is full of garbage:



After one ring latency plus N Token times, all garbage values have been flushed out:



The leader remains stuck until its own rotation counter has filled the ring (at most one more ring latency plus N Token times):

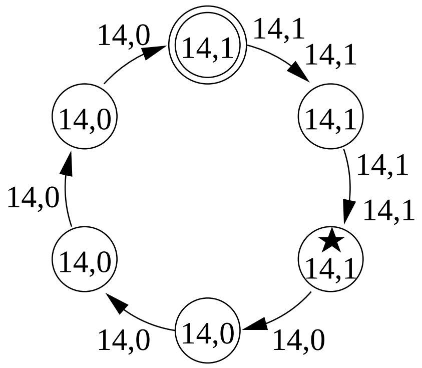


Figure 3.4: Stabilization of the modified FDDI ring.

3.4 Stabilization

A walk-through of the stabilization is shown in Figure 3.4. It is analogous to the stabilization of the theoretical token ring shown in Figure 2.3. A formal proof of stabilization follows.

Start the protocol at time 0 in a state with arbitrary rotation counters and phase bits in the stations, an arbitrary number of stations that think they have a token, and arbitrary Symbols on the ring. We do insist, however, that there be exactly one Leader. If we were using a self-stabilizing Leader election algorithm, that would be guaranteed to become true within a bounded time.

Let n be the number of stations in the ring (at most 1000), let L be the total propagation delay (latency) around the ring (at most 1.617 ms), let TT be the time to transmit a token (1.08 μs), and let $FT_{\max}$ be the time to transmit a maximal Frame (0.36 ms).

At time 0, the Leader may be in the middle of sending a Frame, but within $FT_{\max}$ it will send another message carrying its rotation counter. Thereafter, the Leader regularly emits messages carrying its rotation counter. It includes them in all messages it creates, overwrites them in all messages it repeats, and fills gaps with new Tokens. Imagine that rotation counters (residing in messages or in stations) initially in the ring at time 0 are colored red, except the Leader's, which is green. As time passes, rotation counters that are "descended" from green rotation counters are also green. A green arc will steadily grow, starting at the Leader and proceeding around the ring. When the rotation counter at the frontier of the arc encounters a station, it is copied into the station's internal rotation counter, which turns it green. Then one of the following things happens:

- If the station did not have a token, it repeats the message. It may capture a token in the process, but in any case, the green arc continues to grow with no delay at this station.
- If the station did have a token, and the incoming message caused it to issue a Token, then the green arc continues to grow, with a delay of TT at this station.
- If the station did have a token, and the incoming message did not cause it to issue a Token, then the incoming rotation counter must have matched the

internal rotation counter, which in turn matches the Rotation Counter in the currently outgoing Frame[6], so we might as well consider that Rotation Counter to be green. Hence the green arc continues to grow. The delay at this station might be as much as TT if the outgoing Frame had just barely begun when the incoming rotation counter arrived, or might be less than zero if the outgoing frame was already well underway when the rotation counter arrived.

Let t_2 be the time at which the entire ring becomes green.

$$t_2 \leq FT_{\max} + L + n \cdot TT$$

Let r be the Leader's internal rotation counter at time t_2. The Leader's internal rotation counter has been r ever since time t_1, for some unique $t_1 \leq t_2$.

Imagine that all rotation counters initially in the ring at time 0 are baritone, but at time t_1 the Leader's rotation counter becomes soprano. As time passes, rotation counters that are "descended" from soprano rotation counters are also soprano. A soprano arc will steadily grow, starting at the Leader and proceeding around the ring. The soprano arc has the following properties:

1. All rotation counters within the soprano arc have the value r.
2. The soprano arc contains *all* green rotation counters with the value r.
3. The soprano arc contains at most one token.
4. Within the soprano arc, all phase bits upstream of the token are set, and all phase bits downstream of the token are unset. If the arc contains no token, then all phase bits are set.

At time t_1, when the soprano arc contains only the Leader, the properties must hold. In particular, property 2 holds because the Leader's rotation counter changes only by incrementing, and the counter size (12 bits) was deliberately chosen to be large enough that it cannot wrap around before time t_2.

[6]That is not necessarily true between time 0 and time $FT_{\max}$, but we did not even start growing the green arc until time $FT_{\max}$.

We know that the Leader's rotation counter will not change between times t_1 and t_2, and after that, it cannot change until a Rotation Counter equal to r arrives, which cannot happen until the soprano arc engulfs the entire ring at time t_3. We are therefore assured that until time t_3, the Leader will not introduce any new rotation counter values into the soprano arc that would violate property 1. And because only the Leader is able to generate new rotation counter values, property 2 will remain true as well.

The Leader emits set Phase Bits when it lacks a token, and unset Phase Bits when it has a token, so it respects property 4. Non-Leaders without tokens repeat and copy all Rotation Counters and Phase Bits, so they can't violate any of the properties.

The token-holder within the soprano arc emits only unset Phase Bits, respecting property 4. If it issues a Token, Phase Bits will be set upstream of the Token and unset downstream of the Token, until the token is captured.

The only chance left to violate any of the properties is at the frontier of the arc. When the message at the frontier encounters a station, the station copies the Rotation Counter and Phase Bit, thus joining the arc. There are four possible cases:

- The station had a token, and the message has a set Phase Bit. The station now holds the one and only token in the arc.
- The station did not have a token, and the message has a set Phase Bit. The station may have a token now, in which case it holds the one and only token in the arc.
- The station had a token, and the message has an unset Phase Bit. The station immediately issues a Token.
- The station did not have a token, and the message has an unset Phase Bit. The station still has no token.

In all four cases, properties 3 and 4 are preserved. Summing up, all four properties are preserved until the soprano arc engulfs the entire ring at time t_3. By the same argument used to time the growth of the green arc, we have:

$$t_3 - t_1 \leq L + n \cdot TT$$

Which implies that:

$$t_3 \leq FT_{\max} + 2(L + n \cdot TT) \doteq 5.75\,\mathrm{ms}$$

At time t_3, the properties apply to the entire ring. If there is no token in the ring, the message arriving at the Leader has a set Phase Bit, so the Leader is acquiring a token. Therefore, at time t_3 the ring is in a good state.

Notice that recovery from the most common fault, a corrupted Token that was carrying a token, is very fast—the very next message will deliver the token. The Leader makes sure that the ring is packed tight with messages, so the extra delay incurred will be at most $FT_{\max}$.

3.5 Costs

Our modifications to the FDDI MAC call for longer Tokens (by 5 Symbols). An extra cost of 200 ns is incurred every time a token is captured. In the worst case token rotation, in which every one of 1000 stations captures the token, the added overhead is 200 μs. With Target Token Rotation Times (TTRTs) typically on the order of 10 ms, that is a 2% reduction in usable bandwidth.

We also call for longer Frames (by 6 Symbols). The per-Frame overhead has risen from 56 to 62 Symbols. For very small payloads (32 bytes, which is 64 Symbols), throughput is reduced by 4.8%. For maximum payloads (4478 bytes, which is 8956 Symbols), throughput is reduced by only 0.067%.

We add 27 Symbol times (1.08 μs) to the ring latency as a result of the Leader's buffer, a delay equivalent to 212 m of fiber. The significance of this increased latency depends, of course, on the circumference of the ring.

Another cost is increased complexity in the stations. Each must now include a register to hold a rotation counter and phase bit, plus a 27-Symbol buffer for use when it is the Leader, plus a small amount of extra logic.

3.6 Benefits

The modified protocol recovers from a lost token (the most common fault, usually caused by line noise) faster than the original protocol. The original protocol waits for TVX to timeout (2.5 ms), then reinitializes the ring using the Claim Token process (one ring latency, up to 1.6 ms), which can take as much as 4.1 ms. In the modified protocol, the next message (whether Token or Frame or fragment) arriving after the one that got corrupted will deliver the token, because it carries the same Rotation Counter and Phase Bit that were in the Token before it got corrupted. At the moment the Token was issued, there may have been a partial Frame just upstream of the issuing station (a Frame that the issuing station had previously sent, which had circulated around the ring and was then being stripped). The next message, which might be a Token created by the Leader, or a fragment resulting from another stripped Frame, can therefore lag behind the corrupted Token by as much as one maximal Frame time (0.36 ms). The average lag time will be one half of the average Frame time (0.02 ms for 512-byte payloads).

The original protocol can take *arbitrarily long* to recover from multiple tokens, which can be caused by memory corruption or line noise. The modified protocol can recover from any state[7], including one with multiple tokens, in two rings latencies (up to 3.23 ms) plus two Token times for each station (up to 2.16 ms), plus one maximal Frame time (0.36 ms), a total of at most 5.75 ms.

The Valid Transmission Timer (TVX) is not strictly needed even in the original protocol, but is included because it provides for faster recovery than the Late Counter in most cases (but TVX does not catch as many cases as the Late Counter, see Subsection 3.1.2). The modified protocol already recovers faster than TVX for the most common fault, so it is probably desirable to reduce complexity by eliminating TVX.

3.7 Simulation

We have written a simulator for the modified FDDI MAC which performs all actions down to the Symbol level, except for the following simplifications:

[7] Provided that there is exactly one Leader, see Subsection 3.3.1.

- There are no priorities—stations always use priority 0 for all frames. (This is a mechanism which only affects the timing policies for the capture and release of tokens, and does not concern us.)
- 16-bit addresses are not supported. (In practice, 48-bit addresses are preferred anyway.)
- Ignore_THT and MAC_Reset (conditions imposed from a higher layer) are not supported.
- Frame payloads, also known as Service Data Units (SDUs), are all 0s (we are not interested in the data being carried in the frames).
- The Frame checksum is sent as 0, and not checked upon receipt.
- The 8 redundant bits in the Rotation Counter are sent as 0, and not checked upon receipt. (This only makes the simulation more susceptable to simulated transmission errors.)

We have not conducted an extensive series of tests designed to analyze the performance of the protocol. We have, however, conducted a few tests to check whether the protocol performs roughly as we expect, mainly to provide some confidence that the simulator is correct (and hence, to provide additional assurance that the protocol is correct).

In our test runs, we used a ring of 10 identical stations. Each had a synchronous allocation of 5% of the TTRT, which was 8 ms. Each station had a source of synchronous traffic with an average rate of 2500 kbps, bundled into SDUs whose lengths were exponentially distributed with mean 64 bytes. Each station also had a source of asynchronous traffic with an average rate of 7500 kbps, bundled into SDUs whose lengths were exponentially distributed with mean 256 bytes. The interarrival times of the SDUs were also exponentially distributed (with mean determined by the data rate). Each station addressed its frames to the station exactly opposite it in the ring. Adjacent stations were linked by 1 km of fiber (with a delay of 5.085 ns per meter).

After a Leader had been elected, all the stations had almost all of their state information[8] randomized, and the links were filled with random symbols. In all four trials,

[8]All counters, flags, and registers except those tied intimately to the non-stabilizing Claim Token process (TVX, T_Bid_Rc, T_Neg, T_Opr, T_Req, and Leader_Flag) and those merely concerned with

the ring stabilized in about one ring latency. More precisely, each stabilization began with a small delay (between 0.24 and 1.48 μs) until the Leader issued the first Token, after which the Leader's unique rotation counter value raced around the ring, being copied by all stations along the way, until it arrived back at the Leader, at which point the global state was valid once again. The time for this circumnavigation was exactly one ring latency (51.9 μs, the delay through 10 km of fiber and the Leader's 27 Symbol buffer) in three of the trials, and in the fourth it was longer by just one Token transmission time (1.08μs), because the Rotation Counter value had to push its way through a token holder.

In another series of tests on the same ring configuration, bursty noise was introduced on one of the links to test the recovery time for lost tokens. The time between bursts was distributed exponentially with mean 1 μs, and the length of each burst was distributed exponentially with mean 100 ns (2.5 symbol times). We observed four tokens become lost due to corruption, causing the arrival of the token at the next station to be delayed until the next Frame or Token. The delays ranged from 25 to 54 symbol times (1.0 μs to 2.2 μs). The delays would have been longer if the Frames had been larger; a maximal Frame of 9000 symbols could cause a delay of 360 μs.

token fairness (TRT, THT). Stations were not permitted to jump to states that begin the Claim Token process (CLAIM_TK and TX_BEACON).

Chapter 4

Window Washing

Our attention has been focused on token passing, but observe that the token passing protocol can be used to broadcast a data message (carried in the token packet) reliably from the leader to the other nodes. In fact, if we piggyback data messages on the packets from the leader in a two-node ring, we get a self-stabilizing data link protocol remarkably similar to the famous stop-and-wait data link protocol [20]. The only differences are that the stop-and-wait protocol sends data in the other direction (to the leader), and lacks a minimum modulus requirement.

Unfortunately, using token-passing to send data requires each message to be delivered and acknowledged before the next message can be sent. Whereas the early work on counter flushing [22] applied to the problem of broadcasting a *single* message at a time to a set of receivers, we have generalized the approach to allow *pipelining*. Section 4.1 starts the chapter with the simplest pipelined protocol, the non-stabilizing two-node sliding window protocol. Section 4.2 derives window washing to provide stabilization, which is proven correct in Section 4.3. Section 4.4 shows how to allow multiple receivers. Finally, Section 4.5 adapts the new protocol to the problem of flow-control. All protocols discussed in this chapter assume the network model defined in Section 2.1.

Figure 4.1 shows graphically how the multitude of protocols mentioned in this chapter relate to one another.

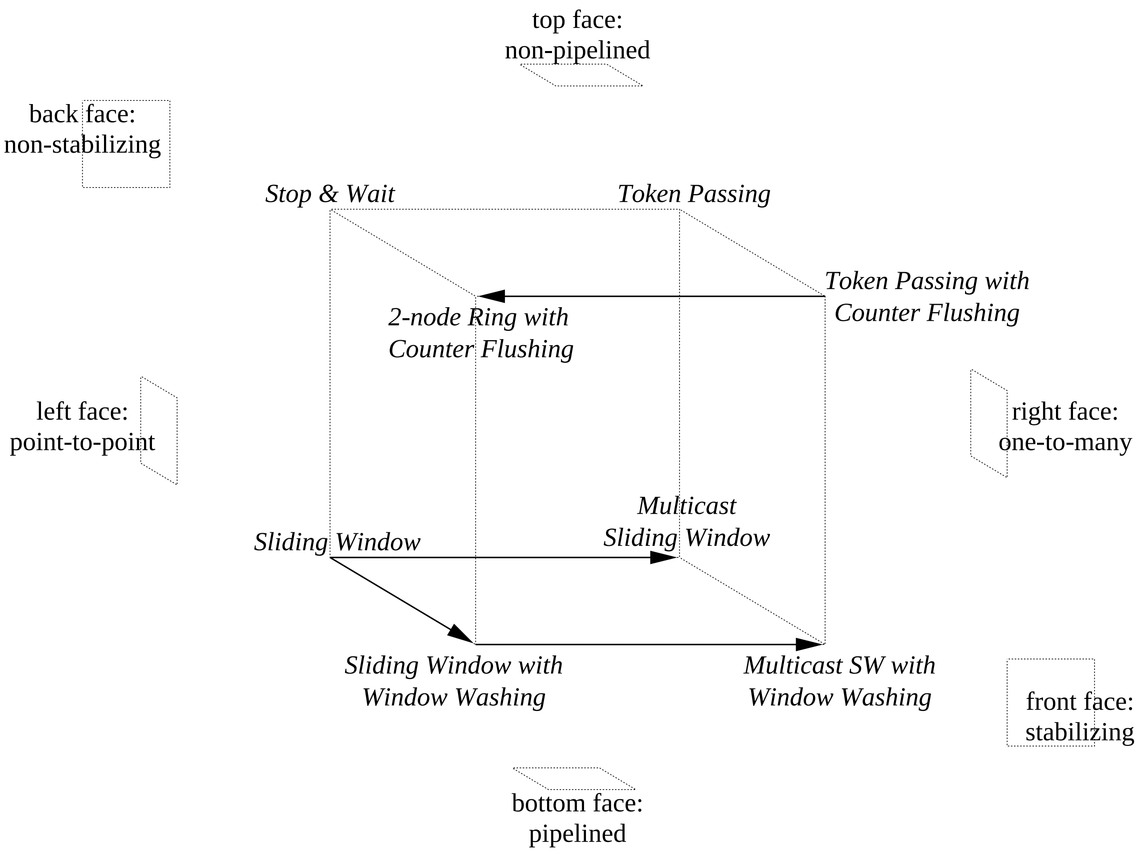


Figure 4.1: Organization of protocols (vertices) and derivations (arrows) mentioned in Chapter 4.

4.1 Sliding Window

A sliding window protocol [20] is a two-node protocol enabling a sender to send a sequence of messages to a receiver without duplication, loss, or misordering. The sender attaches a sequence number seq to each message m to form a data packet, periodically retransmitting until an acknowledgement ("ack") arrives indicating that the message has been received. To allow pipelining a window of up to w messages before receiving an ack, the sender keeps a lower window edge L. The sender may send packets only with sequence numbers in the range $[L+1, L+w]$, where w is the window size.

All arithmetic is modulo M where $M > w$.

SendData(seq, m) (* Sender emits data packet. *)
  Preconditions:
    $seq \in [L+1, L+w]$
    m is the message associated with seq

ReceiveData(seq, m) (* Receiver absorbs data packet. *)
  Effects:
    if $seq = R+1$ then
      $R \leftarrow seq$
      deliver message m

SendAck(ack) (* Receiver emits ack. *)
  Preconditions:
    $ack = R$

ReceiveAck(ack) (* Sender absorbs ack. *)
  Effects:
    $L \leftarrow ack$

Figure 4.2: Sliding window code.

The receiver keeps a receive sequence number R (initially 0) that records the last sequence number it accepted. For simplicity, we consider a sliding window protocol which neither buffers out-of-order messages nor does selective reject [20]. That is, the receiver accepts a packet only if $seq = R+1$, at which point the receiver copies seq into R, otherwise it discards the packet. The sender periodically retransmits packets in its window, and the receiver periodically sends an ack packet containing the value $ack = R$. All arithmetic on sequence numbers is assumed to be modulo M, the size of the bounded integer space. Typically M is required to be greater than w [20]. However, for self-stabilization we will require a larger value. The code for this protocol, sometimes known as "Go Back N", is shown in Figure 4.2. We have not modeled the arrival of messages from a higher layer to the sender—for simplicity, we assume that the sender generates a unique message for each sequence number.

Our abstract version of a sliding window protocol does reflect real protocols. Real sliding window protocols sometimes send an ack whenever a new packet is accepted;

our protocol can simulate such behaviors. Although in our protocol it appears that retransmissions of packets are constantly enabled, this merely ensures that the protocol works correctly regardless of timer values. Our abstract protocol must be augmented by sensible retransmission policies (especially the choice of retransmission timer value) to provide good *performance* when implemented in a real system.

4.2 Derivation

It turns out that merely increasing the modulus M to exceed $w \cdot c_{max}$ will make the sliding window code self-stabilizing. However, there would be a large problem: the receiver is leading. The receiver is very particular about what *seq* value it will accept, whereas the sender always copies incoming *ack* values into L. Therefore, if the protocol ever entered a bad state, the sender would resync to the receiver. But the data messages originate at the sender, and are numbered by the sender, so the result might be that the sender would have to resend long-forgotten messages, or discard huge numbers of messages. Also, we will later wish to increase the number of receivers, but it makes no sense to have more than one leader. The first change we must make, therefore, is to have the sender lead.

Sender Ack Check In correct operation, the acks arriving at the sender are always in the range $[L, L+w]$. Let the sender ignore acks outside this range, thereby asserting its correctness (additionally, it might as well ignore acks equal to L, since copying them would have no effect). This is called the *sender ack check*. It is analogous to the leader in counter flushing acting on only one particular value, but in this case, there is a range of w values on which the leader will act.

Receiver Window Adjustment Of course, unless we make the receiver follow, a bad state could leave the system in deadlock. If L and R differ by more than w, then both the sender and receiver will stubbornly ignore packets sent by the other. In correct operation, if the sender were to include a copy L' of L in each data packet, the receiver would find, whenever it received a packet, that R is in the range $[L', L'+w]$. Let the sender actually include L' in the packets, and let the receiver, whenever it receives a packet which does not include R in that range, resync by setting R to *seq* from the packet. This is called the *receiver window adjustment.*

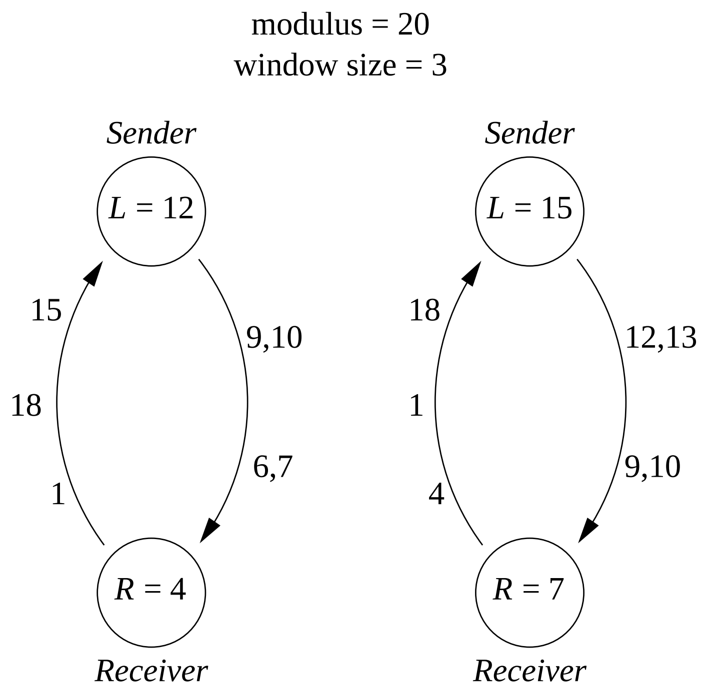


Figure 4.3: Livelock caused by a too-small modulus. The numbers on the downward links are L', seq pairs. One execution of each action transforms the state on the left into the state on the right, which is identical except that every value has increased by the window size 3.

Now that we have the sender leading and the receiver following, we are as close to having a self-stabilizing protocol as we were to begin with. There still remains, however, the problem of the size of the modulus M. If M is too small, aliasing can allow a bad state to leave the system in livelock, with the receiver continually adjusting but never catching up to the sender. For example, suppose that the data packets on the downward (sender-to-receiver) link, from newest to oldest, have L' values of $L-w, L-2w, \ldots$ and *seq* values of $L+1-w, L+1-2w, \ldots$ Further suppose that the receiver number R is w less than the oldest *seq* value on the downward link, and that the *ack* values on the upward (receiver-to-sender) link, from newest to oldest, are $R-w, R-2w, \ldots$ Finally, suppose that M is just small enough that the oldest *ack* value on the upward link equals $L+w$. The sender may emit a packet with $seq = L+1$, then absorb the ack with value $L+w$, setting L to $L+w$. The receiver may at the same time emit an ack with value R, then absorb the packet with $seq = R + w$, adjusting the out-of-range R to $R + w$. Notice that the state now looks just as it did at the beginning, except that all the values have increased by w. This scenario, depicted in Figure 4.3, could continue indefinitely.

Minimum Modulus The livelock scenario depends on the ability to place a sequence of numbers on the ring spaced w apart that wrap around the bounded integer space. As before with counter flushing, let c_{max} be the maximum number of counter values (L, seq, R, and ack values, but not L' values) that may be stored in the links and nodes. By requiring $M \geq w \cdot c_{max}$ we foil the scenario described. However, the requirement $M > w \cdot c_{max}$ will make the proof easier.

All arithmetic is modulo M where $M > \boxed{w \cdot c_{max}}$.

SendData($\boxed{L',}\, seq, m$) (* Sender emits data packet. *)
  Preconditions:
    $\boxed{L' = L}$
    $seq \in [L + 1, L + w]$
    m is the message associated with seq

ReceiveData($\boxed{L',}\, seq, m$) (* Receiver absorbs data packet. *)
  Effects:
    if $\boxed{R \notin [L', L' + w] \text{ or}}$ $seq = R + 1$ then
      $R \leftarrow seq$
      deliver message m

SendAck(ack) (* Receiver emits ack. *)
  Preconditions:
    $ack = R$

ReceiveAck(ack) (* Sender absorbs ack. *)
  Effects:
    $\boxed{\text{if } ack \in [L + 1, L + w] \text{ then}}$
      $L \leftarrow ack$

Figure 4.4: Stabilizing sliding window code. Changes from the original code are boxed.

Table 4.1: The generalization from counter flushing to window washing.

	counter flushing	window washing
mechanisms	minimum modulus leader counter check	minimum modulus sender ack check window adjustment
number of values leader will accept	1	w
minimum modulus	Max	$w \cdot Max$
stabilization time	2 round-trips	2 round-trips

The three mechanisms—sender ack check, receiver window adjustment, and minimum modulus—are together called *window washing*. The modifications to the sliding window code are shown in Figure 4.4. Note that none of the new code gets used in correct operation—it comes into play only when the system is in a bad state. Table 4.1 summarizes the generalization from counter flushing to window washing.

4.3 Correctness

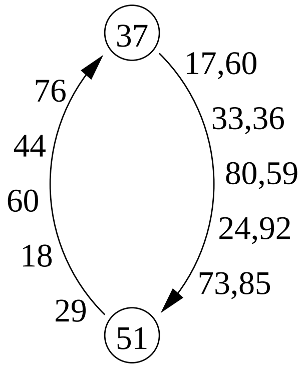
The stabilization of the protocol is exactly analagous to the stabilization of the token ring using counter flushing. Once a data packet has traversed the downward link and an ack has subsequently traversed the upward link (one round-trip delay), all initial garbage values have been flushed from the system, and the leader has advanced, but not far enough to have wrapped all the way around the bounded integer space. Therefore, the sender ack check will keep the leader stuck until the rest of the system has caught up (a second round-trip delay). An example is shown in Figure 4.5. The following proof of correctness refers to the actions of Figure 4.4. All arithmetic is modulo M, and $M > w$.

4.3.1 Definitions

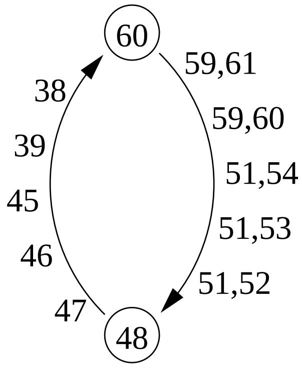
In a given state:

- The *upward sequence* is the sequence of values starting with R and proceeding through all the in-transit *ack* values in the reverse link. (The author of this

The sender is the top node. The window size is $w = 10$. In the initial state, the system is full of garbage:



After one round-trip delay, all garbage values have been flushed out:



The sender remains stuck until the receiver has caught up (at most one more round-trip delay):

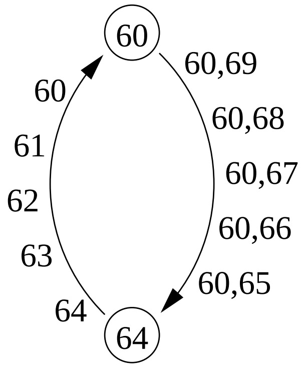


Figure 4.5: Stabilization of the sliding window using window washing.

proof visualizes the sender as being above the receiver, hence the name of the sequence.)

- The *downward sequence* is the sequence of values starting with L and proceeding through all the L' values in the in-transit data packets in the forward link.

- The *validity sequence* is the concatenation of the upward and downward sequences, in that order.

A sequence of bounded integers is *non-increasing* iff each successive value is either equal to the previous value, or does not occur in the range $[p, f]$, where p is the previous value and f is the first value. For example, $\langle 4, 3, 3, 0, 99, 43, 6\rangle$ is non-increasing, and so is $\langle 4, 6\rangle$, but $\langle 4, 5, 6\rangle$ is not, because 6 *is* in the range $[5, 4]$ in modular arithmetic.

Define a *valid state* as one that meets the following conditions:

1. For every data packet in transit,
   $seq \in [L' + 1, L' + w]$.

2. The validity sequence is non-increasing.

3. The validity sequence spans a range of at most w, which is to say, every value in the sequence is in $[R - w, R]$. Equivalently, every value in the sequence is in $[L^*, L^* + w]$, where L^* is the last value in the sequence.

4.3.2 Safety and Liveness

Conditions 2 and 3 together imply that every value in the upward sequence is in $[L, L + w]$. Therefore, the sender ack check has no effect on a valid state.

Condition 3 implies that $R \in [L', L' + w]$ for any data packet arriving at the receiver. Therefore, the receiver window adjustment has no effect on a valid state.

Valid states are *stable*. That is, any of the four actions of the protocol, applied to a valid state, produces a valid state, as shown here:

Condition 1 is obviously preserved by SendData, and unaffected by all other actions.

Conditions 2 and 3 are clearly preserved by both SendData and SendAck, because they just duplicate a value in the validity sequence. They are clearly preserved by ReceiveAck, because it just deletes a value from the sequence.

In ReceiveData, a value is deleted from the end of the sequence, which is safe. But the first value in the validity sequence, R, can also change. Because the receiver window adjustment does not apply, R can change only if $seq = R + 1$, in which case R increments. Incrementing the first value in a non-increasing sequence can break the non-increasing property only if it causes the sequence to wrap completely around the bounded integer space. But using condition 1, we see that condition 3 is preserved, so the sequence could not possibly wrap around, so condition 2 is preserved. □

Valid states may be stable, but do they deserve to be called "valid"? Yes. Notice that starting from any valid state, data packets are accepted in order, because they are accepted only when $seq = R+1$, which causes R to increment. Also note that liveness is guaranteed: if L tries to get stuck, the sender will eventually send a data packet with $L' = L$ and $seq = L + 1$, which will cause R to enter the range $[L + 1, L + w]$ if it was not already in that range. R cannot leave that range as long as L is stuck. The receiver will eventually send an ack in that range, which will cause L to advance. Thus, L can remain stuck for at most one round-trip delay, plus the time to transmit a window of data.

The claim of liveness above, and the time bounds on self-stabilization that will be derived below, both depend on data always being available to send. If this is not actually the case, but we still want time bounds, we can allow the sender, whenever its message queue becomes empty, to enqueue a null message that does not affect the data stream. Artificial delays can accompany such null messages so that they do not consume too much bandwidth (a null message must still carry L' and seq). Alternately, we may not care about liveness and stabilization while there is no data to send.

4.3.3 Stabilization

We need to show that an arbitrary state will reach a valid state. Call the initial state $\mathcal{S}_0$.

Consider the first data packet sent. Wait until it arrives at the receiver, producing state $\mathcal{S}_1$. Note that condition 1 is true in $\mathcal{S}_1$, and that the code for SendData insures that it will stay true thereafter. Now wait until all acks in the reverse link in state $\mathcal{S}_1$ have arrived at the sender, producing state $\mathcal{S}_2$. Let L_0 and L_2 be the sender's L values in states $\mathcal{S}_0$ and $\mathcal{S}_2$.

How many distinct *ack* values could the sender have received during this time? Notice that the receiver can change R only when it receives a packet. The answer, therefore, is no more than one for each ack in the reverse link in state $\mathcal{S}_0$, plus one for the R value in state $\mathcal{S}_0$, plus one for each packet in the forward link in state $\mathcal{S}_0$. The maximum possible value of this sum is $c_{max} - 1$. (Recall that c_{max} is the maximum number of data packets, acks, and nodes in the system.)

Let us assume that the modulus M is larger than $w \cdot c_{max}$.

Since the sender can change L only when it receives an ack, and can advance L by at most w each time, L could have advanced by at most $w(c_{max} - 1)$ between states $\mathcal{S}_0$ and $\mathcal{S}_2$. Note that ever since state $\mathcal{S}_1$, every L' value in the forward link has been in $[L_0, L_2]$, and every *seq* value has been in $[L_0 + 1, L_2 + w]$. Because the receiver, upon receiving a packet, must set R to *seq* unless R is in the range $[L', L' + w]$, we know that R has been in the range $[L_0, L_2 + w]$ ever since state $\mathcal{S}_1$, which means that in state $\mathcal{S}_2$, all *ack* values in the reverse link are in $[L_0, L_2 + w]$. Thus, in state $\mathcal{S}_2$, every value in the system is in the range $[L_0, L_2 + w]$.

Because the modulus M is larger than $w \cdot c_{max}$, we can define a total ordering on the values in the range $[L_0, L_2 + w]$, with L_0 as the "zero" value, and each successive value being the next larger value.

We can now use words like "increase" and "decrease" to describe variables that stay in this range. For instance, during the time interval in question, from state $\mathcal{S}_0$ to state $\mathcal{S}_2$, L never decreases. Hence the downward sequence in state $\mathcal{S}_1$ is non-increasing. In state $\mathcal{S}_1$, the receiver has just received a data packet containing some L' value, which means that R must be in the range $[L', L' + w]$ (whether it accepted the packet or not—see the ReceiveData code). Because each successive packet that it receives until state $\mathcal{S}_2$ contains an L' value larger than (or equal to) the previous one, R is always $\leq L^* + w$, where L^* is the last value in the downward sequence. Therefore, subsequent changes to R can only be increases, never decreases. Consequently, in state $\mathcal{S}_2$, the upward sequence is non-increasing.

$$time(\mathcal{S}_2) - time(\mathcal{S}_0) \leq \textit{round-trip delay}$$

What happens after $\mathcal{S}_2$? L will remain stuck at L_2 until an *ack* value arrives in the range $[L_2 + 1, L_2 + w]$. Until that happens, we preserve the property that all values in the system are in the range $[L_0, L_2 + w]$, and the non-increasing property of the downward and upward sequences, and therefore we preserve the property that $R \leq L^* + w$. Denote by $\mathcal{S}_3$ the first state after $\mathcal{S}_2$ (or equal to $\mathcal{S}_2$) in which the smallest (i.e. oldest) *ack* value in the reverse link is in the range $[L_2, L_2 + w]$. This is a prerequisite for an ack in the range $[L_2 + 1, L_2 + w]$ to arrive at the sender, so we know that the properties mentioned above have been preserved through state $\mathcal{S}_3$. We also know that we must reach $\mathcal{S}_3$. Why? Suppose we never do. Then L will remain at L_2 forever. A data packet with $L' = L_2$ will be sent. After it arrives at the receiver, R must be in the range $[L_2, L_2 + w]$. R never decreases, and stays in the range $[L_0, L_2 + w]$, so it must stay in the range $[L_2, L_2 + w]$. It will send an ack in this range, which will make its way toward the sender until all older acks have been flushed out. At this point, we have reached $\mathcal{S}_3$.

$$time(\mathcal{S}_3) - time(\mathcal{S}_2) \leq \textit{round-trip delay}$$

Examine $\mathcal{S}_3$. The downward and upward sequences are non-increasing. The last value in the upward sequence (the smallest *ack* value) is $\geq L$, so condition 2 is satisfied. $R \leq L^* + w$, so condition 3 is satisfied. Condition 1 has been satisfied since state $\mathcal{S}_1$. $\mathcal{S}_3$ is therefore a valid state. It has taken at most two round-trip delays to get there from state $\mathcal{S}_0$.

4.4 Multiple Receivers

In normal operation, the sliding window protocol guarantees the in-order acceptance of a sequence of packets from a single sender by a single receiver. A one-to-many sliding window protocol would make the same guarantee for each of several receivers, all accepting the same sequence of packets from the sender. This problem is particularly interesting in the context of modern networks (like ATM and the Internet) where a sender can multicast values to many receivers. However, most current protocols[1] *do not* offer reliable one-to-many transmission—packets can be lost. Reliable multicast can simplify the design of many real applications, including database update and distributed simulation. The upper layer would still provide its own end-to-end reliability, but it need not trouble itself with performance optimizations, because incorrect behavior in the self-stabilizing lower layer will be extremely rare and short-lived.

We now describe a stabilizing one-to-many sliding window protocol that would be technically feasible to implement in ATM switches and IP routers, although the large installed base makes implementation economically infeasible.

4.4.1 Architecture

The trivial way to build a one-to-many sliding window protocol would be simply to run several sliding window protocols concurrently, with a separate communication path between the sender and each receiver. But to ease the load on the sender and to reduce traffic, the scheme should take advantage of the topology of the receivers, and allow them to relay packets from the sender to other nearby receivers.

Despite the fact that some receivers may not get packets directly from the sender, our general strategy is to have the protocol simulate the behavior of multiple two-node protocols between the sender (node 0) and each receiver i. By doing so, we can leverage off the design and proof of the two-node sliding window protocol. To abstract the notion that a receiver i gets packets from the sender indirectly through other relay nodes, we define the notions of a *data tree* and an *ack tree*, which are independent spanning trees[2] of the network, both rooted at the sender. The data tree carries data packets down from the sender to the receivers, while the ack tree carries acks up from the receivers to the sender. The parent of each node i in the data tree is the node before i on the path from the sender to i; it relays packets to i from the sender. Similarly, the parent of node i in the ack tree is the node after i on the path from i to the sender; it relays acks from i toward the sender. When the

[1] A few recent Internet protocols like SRM [5] offer reliable multicast, but these protocols have not been proven to be self-stabilizing.

[2] We could generalize these notions to acyclic graphs, but that would clutter the presentation.

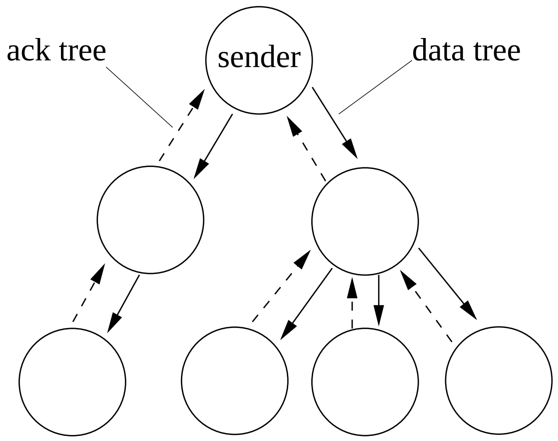


Figure 4.6: The data and ack trees for a tree topology.

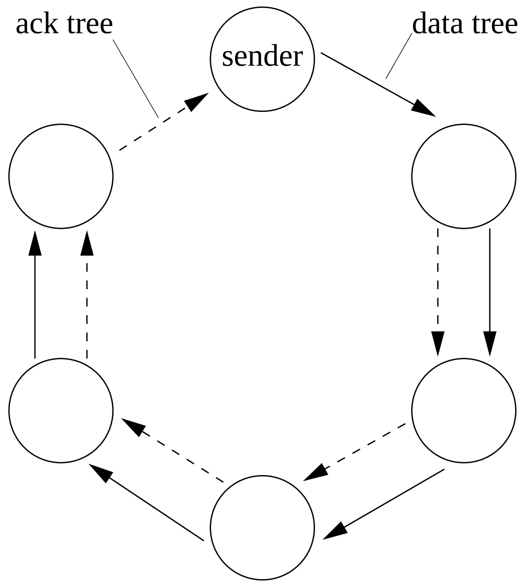


Figure 4.7: The data and ack trees for a clockwise ring topology.

physical topology of the network is a tree (with the sender at the root), the data tree contains all the leafward links, and the ack tree contains all the rootward links, as shown in Figure 4.6. But arbitrary physical topologies are permissible; for example, Figure 4.7 shows the data and ack trees for a ring topology.

4.4.2 Protocol

The suggested one-to-many architecture could be used to make straightforward alterations to the non-stabilizing sliding window protocol, resulting in a non-stabilizing one-to-many sliding window protocol. While this might be interesting in its own right, we will focus our attention on the stabilizing sliding window protocol, and the resultant stabilizing one-to-many sliding window protocol.

Each receiver behaves as in the two-node protocol, except that it forwards a copy of every incoming data packet to its children in the data tree, and it handles acks slightly differently. An ack packet now contains not just one *ack* value, but one for each node below it in the ack tree. Each node (including the sender) stores the last ack to arrive from each of its children in the ack tree. Each receiver includes the contents of those stored acks, as well as its own R value, in each ack it sends upward.

The sender also behaves somewhat differently. Whenever an ack arrives, the sender stores it and then, if and only if *every one* of the *ack* values in *each* of its stored acks is in the range $[L+1, L+w]$, it advances L to the *nearest* of those *ack* values. The code for the one-to-many protocol appears in Figure 4.8.

4.4.3 Avoiding Implosion

It appears that we have a problem with implosion. If a node has n descendents in the ack tree, then it needs $O(n)$ space to store the acks and $O(n)$ bandwidth to receive them. It also takes $O(n)$ time for a receiver to construct an ack. However, it is possible to reduce the space and bandwidth requirements to $O(1)$, and the time requirement to $O(k)$, where k is the degree of the node.

All arithmetic is modulo M where $M > w \cdot c_{max}$.

$\mathsf{SendData}_{0j}(L', seq, m)$ (* Sender emits data packet to datachild j. *)
Preconditions:
$L' = L$
$seq \in [L+1, L+w]$
m is the message associated with seq

$\mathsf{ReceiveData}_i(L', seq, m)$ (* Receiver i absorbs data packet from dataparent. *)
Effects:
if $R_i \notin [L', L'+w]$ or $seq = R+1$ then
$R \leftarrow seq$
deliver message m
for each datachild j enqueue $\langle L', seq, m\rangle$ onto Q_j

$\mathsf{SendData}_{ij}(L', seq, m)$ (* Receiver i emits data packet to datachild j. *)
Preconditions:
$\langle L', seq, m\rangle$ is at the head of Q_j
Effects:
dequeue one packet from Q_j

$\mathsf{SendAck}_i(A)$ (* Receiver i emits ack to ackparent. *)
Preconditions:
A is the union of $\{\langle i, R_i\rangle\}$ and A_j, for each ackchild j

$\mathsf{ReceiveAck}_{ji}(A)$ (* Receiver i absorbs ack from ackchild j. *)
Effects:
$A_j \leftarrow A$

$\mathsf{ReceiveAck}_{j0}(A)$ (* Sender absorbs ack from ackchild j. *)
Effects:
$A_j \leftarrow A$
if for each ackchild j, for each $\langle i, R\rangle \in A_j$, $R \in [L+1, L+w]$ then
$L \leftarrow$ the earliest of all those R values

Figure 4.8: Stabilizing one-to-many sliding window code. Every receiver contains a queue Q_j for each of its children in the data tree. Every node contains a set A_j for each of its children in the ack tree.

Suppose, instead of transmitting and storing a whole set of *ack* values, we instead use a fixed-size *digest* that holds either a single pair $\langle lo, hi\rangle$, or the special value wide. A nonempty set x maps to a digest $f(x)$ as follows.

If all the *ack* values in x lie within a range of *MaxWidth* consecutive values, then $f(x) = \langle lo, hi\rangle$, where lo is the earliest *ack* value, and hi is the latest. Otherwise, $f(x) =$ wide. Notice that the terms "earliest" and "latest" make sense only if *MaxWidth* is no more than half the modulus of the bounded integers. *MaxWidth* will be further constrained later.

Let $x = x_1 \cup x_2 \cup x_3 \cup \ldots \cup x_k$. $f(x)$ can be computed from $f(x_1)$, $f(x_2)$, $f(x_3)$, …, $f(x_k)$ as follows. If any of the $f(x_i)$ is wide, then $f(x) =$ wide. Otherwise, we have $f(x_i) = \langle lo_i, hi_i\rangle$. If all the lo_i and hi_i values lie within *MaxWidth* consecutive values, then $f(x) = \langle lo, hi\rangle$, where lo is the earliest of those values, and hi is the latest. Otherwise, $f(x) =$ wide.

If $MaxWidth \geq w$, we can see from the definition of $f(x)$ that the *ack* values in a set x are all within $[L+1, L+w]$ if and only if $f(x) \neq$ wide, and $f(x) = \langle lo, hi\rangle$, and lo and hi are both in $[L+1, L+w]$. Furthermore, if all the *ack* values are in $[L+1, L+w]$, then the earliest of them is lo.

We have now converted all operations that the protocol actions perform on sets of *ack* values into corresponding operations on digests. The replacement code is shown in Figure 4.9.

One last note: choosing $MaxWidth > w$ (rather than merely $MaxWidth \geq w$ as required above) provides the unnecessary but appealing property that no wide values appear anywhere in any valid state.

4.4.4 Stabilization

Intuitively, each two-node protocol within the one-to-many protocol behaves exactly as before, except that its progress is held hostage by the progress of the slowest two-node protocol. However, this does not complicate matters significantly. Since each of the two-node protocols stabilizes independently, we can reuse most of the proof of stabilization of the two-node protocol. The liveness argument changes, since it must now involve global reasoning about all the receivers. The formal proof of stabilization

$\mathsf{SendAck}_i(A)$ (* Receiver i emits ack to ackparent. *)
  Preconditions:
    if for any ackchild j, $A_j = \mathsf{wide}$ then
      $A = \mathsf{wide}$
    else
      $T = \{R_i, lo, hi \mid \langle lo, hi \rangle = A_j$ for any ackchild $j\}$
      if all members of T lie within $w + 1$ consecutive values then
        $A = \langle$earliest member of T, latest member of $T\rangle$
      else
        $A = \mathsf{wide}$

$\mathsf{ReceiveAck}_{0j}(A)$ (* Sender absorbs ack from ackchild j. *)
  Effects:
    $A_j \leftarrow A$
    if $A_j \neq \mathsf{wide}$ for each ackchild j then
      $T \leftarrow \{lo, hi \mid \langle lo, hi \rangle = A_j$ for any ackchild $j\}$
      if all members of T are in $[L + 1, L + w]$ then
        $L \leftarrow$ earliest member of T

Figure 4.9: Replacement stabilizing one-to-many sliding window code using ack digests.

for the one-to-many protocol is relegated to Appendix B, because it is so similar to the proof for the two-node protocol.

4.5 Flow Control

Let us return to the two-node stabilizing sliding window protocol. For simplicity, we have been ignoring the possibility that packet *acceptance* and packet *delivery* (up to the next higher layer) at the receiver happen at different times. In reality, there is probably a queue Q into which packets are enqueued when they are accepted, and from which they are dequeued just before they are delivered. We may wish the sliding window protocol to prevent the queue from overflowing in normal operation; that is, we may wish to add *flow control* to the protocol.

This is a very easy modification. Simply have the receiver, instead of sending acks with the value R, send acks with the value $R - length(Q)$. This can be easily accomodated in the proof of stabilization, and ensures that the queue never contains more than w packets in normal operation. The receiver is therefore free to limit the size of the queue to w, and drop packets that do not fit.

In some cases, such as flow-controlled virtual circuits between ATM switches, we may be interested only in flow control, and not in reliability (higher layers may be taking care of that). We can then simplify the protocol as follows.

Receiver Window Adjustment In the reliable protocol, the receiver accepts a packet if it is the expected one, or if its L' value indicates that the receiver is out of sync; otherwise the receiver rejects the packet. If we do not care about reliability at this layer, there is no need ever to reject packets—the receiver can simply always accept the packet. There is no need for packets to carry L' values.

Sender Ack Check Since we are no longer providing reliability, the sender need not retransmit packets. But we would still like packets to arrive in order in the absence of errors. Since the receiver is accepting all packets, we will have the sender send them strictly in order. We will therefore need a second counter at the sender, S, which holds the last sequence number sent. With the existing ack check in place (the sender ignores *ack* values outside the range $[L+1, L+w]$), the sender may send a packet whenever $S \neq L+w$. However, we may simplify matters by combining these two checks into one: let the sender copy *any* ack value into L, but let it send a packet only when $S \in [L, L+w-1]$. While the new check is not precisely equivalent to the old checks, it serves the same purpose. The purpose of the original ack check was to prevent a garbage ack from causing a large jump in the sequence numbers coming out of the sender. With the new check, a garbage value in L will stop the sender from sending anything until a plausible ack arrives.

Minimum Modulus We have required the modulus M to be greater than $w \cdot c_{max}$ because L could jump by w for each ack received. This is still true, but now we have the additional restriction that S can jump by at most 1 for each packet sent. This allows us to tighten the bound on M by about a factor of 2. The livelock scenario analogous to the one in Figure 4.3 is shown in Figure 4.10. It is foiled if $M > w \cdot r_{max} + f_{max}$, where r_{max} is the maximum number of acks that fit in the

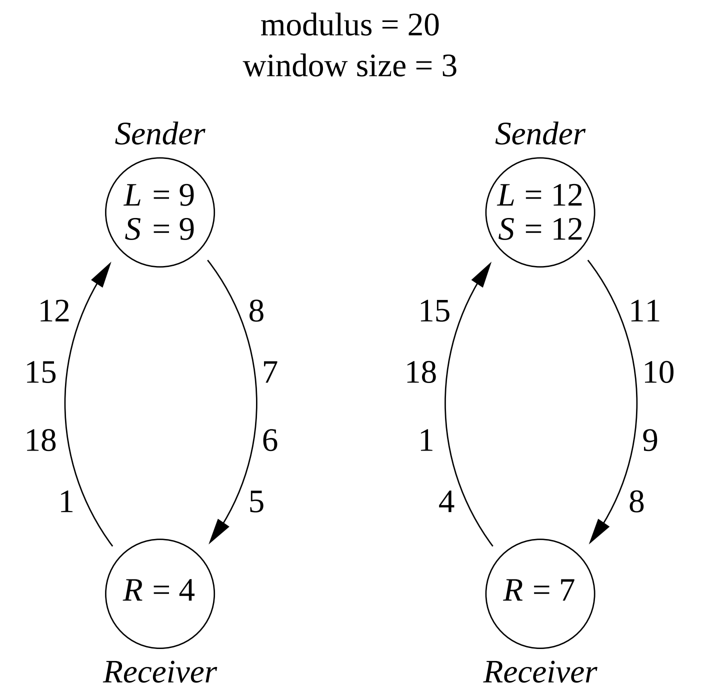


Figure 4.10: Livelock caused by a too-small modulus. w executions of SendData and ReceiveData plus one execution of SendAck and ReceiveAck transform the state on the left into the state on the right, which is identical except that every value has increased by the window size 3.

reverse link plus one, and f_{max} is the maximum number of data packets that fit in the forward link plus one.[3]

Finally, one last simplification we can make is to decouple sequence numbers from data messages. Since we care only about flow control, not reliability, we do not need to keep track of which sequence number belongs to which message—we are concerned only with how many data packets are sent, received, enqueued, etc. Therefore, not every data packet must carry a sequence number. Instead, sequence numbers may be sent occasionally in special sync packets. This is equivalent to having the receiver assume

[3]If we do not mind talking about time, we can further relax the restriction to

$$M > \frac{\textit{max round-trip delay}}{\textit{min packet transmission time}} + w$$

which is enough to insure that, during the first round-trip delay, S will not advance too far, and will therefore be "stuck" until the receiver catches up.

that unmarked data packets would have contained the expected sequence number, and checking this assumption whenever a sync packet arrives. The separation of data packets and sync packets is especially useful when the packets are ATM cells, which lack room for sequence numbers.

All arithmetic is modulo M, where $M > w \cdot r_{max} + f_{max}$

$\mathsf{SendData}(m)$ (* Sender emits data packet. *)
  Preconditions:
    $S \in [L, L + w - 1]$
  Effects:
    $S \leftarrow S + 1$

$\mathsf{SendSync}(seq)$ (* Sender emits sync packet. *)
  Preconditions:
    $seq = S$

$\mathsf{ReceiveData}(m)$ (* Receiver absorbs data packet. *)
  Effects:
    $R \leftarrow R + 1$
    if Q is not full then enqueue message m onto Q

$\mathsf{ReceiveSync}(seq)$ (* Receiver absorbs sync packet. *)
  Effects:
    $R \leftarrow seq$

$\mathsf{SendAck}(ack)$ (* Receiver emits ack. *)
  Preconditions:
    $ack = R - length(Q)$

$\mathsf{ReceiveAck}(ack)$ (* Sender absorbs ack. *)
  Effects:
    $L \leftarrow ack$

Figure 4.11: Stabilizing sliding window code for flow-control without reliability.

The simplified code is shown in Figure 4.11. It is not new—it is virtually identical to the Credit Update Protocol (CUP) [9] proposed for ATM flow control and implemented in Bell-Northern Research switches. But now that we can see that it uses window washing, we know that it recovers from arbitrary faults in two round-trip delays, *provided that the bounded integers use a large enough modulus.* (Actually, we must also require that the queue be incapable of holding more than a limited number of packets, lest memory corruption cause the next higher layer to receive garbage packets for a long time.)

There is one major difference between CUP and the protocol of Figure 4.11: the latter assumes a cheap $length(Q)$ function. CUP never counts the occupancy of the queue (presumably because it is expensive in many real queue implementations). Instead, the receiver keeps a variable D (conceptually, the number of messages delivered or dropped), and maintains the invariant $D = R - length(Q)$ by updating D whenever R or Q changes. CUP cannot, therefore, stabilize from a state in which the invariant does not hold (thus, it requires initialization, and cannot survive memory corruption at the receiver). It may be worth investigating the feasibility of queue implementations for which the $length()$ function is cheap. For example, if the queue is an array of n pointers to buffers, plus two indices for the head and tail, the $length()$ function is simply a subtraction of the indices modulo n. This simple scheme does not allow the capacity of the queue to vary, but more versatile implementations may be possible.

Chapter 5

Conclusions

We have built upon the foundation of counter flushing in two directions. First, we have demonstrated that self-stabilizing protocols are feasible in real networks. Our modified FDDI MAC token-passing protocol requires fairly minor changes to the existing specification. The advantages—increased robustness, like the ability to recover from multiple tokens, and faster recovery from the most common fault, a lost token—outweigh the costs of slightly reduced throughput and slightly increased complexity.

In the other direction, we have expanded the theory, devising a bounded integer sliding window protocol that is simpler, faster, and based on a more general mechanism than Spinelli's earlier one. We have generalized sliding window protocols to the multiple receiver case and shown that our stabilizing techniques apply there as well. Additionally, we have provided evidence that the window washing mechanism is versatile, by applying it to the problem of flow control.

We have tried to pay attention to both theory and practice. In the process of applying counter flushing to FDDI, we learned to solve problems that do not arise in the abstract model. For example, a real ring has both a ring latency (short) and a token rotation time (longer), two very distinct concepts, whereas the abstract model had just one notion of round-trip delay. This dichotomy led us to split the counter value into a separate rotation counter and phase bit, so that the stabilization time would be proportional to the short ring latency. In the abstract model stations retransmit their counter values, but in a real ring this would require extra delay at every station, which led us to consolidate the burden of retransmission at just one station.

Our work on generalized sliding window protocols is more theoretical, but even there we have paid attention to practical issues, like queue implementations for the flow control protocol, and memory and bandwidth requirements in the one-to-many sliding window. In fact, with no requirement for the internal nodes to do retransmission, the reliable multicast protocol appears simple enough to implement in ATM switches. In general, our goal has been to bridge the gap between elegant theoretical techniques for fault-tolerance and the needs of real networks.

Appendix A

Changes to the FDDI MAC Specification (ANSI X3.139-1987)

Remove all references to restricted tokens.

4.2.1 Token.

After FC add RC (Rotation Counter) (12 bits encoded in 5 Data Symbols for corruption detection) and PB (Phase Bit) (1 bit encoded in 1 Data Symbol for corruption detection).

Replace ED (End Delimiter) with an Idle Symbol, so that a Token looks like a Frame fragment (which is how it is treated).

4.2.2 Frame

Add RC and PB as in 4.2.1.

4.3.7.1 Token Ending Delimiter.

Remove this section.

4.3.9 Rotation Counter and Phase Bit.

New section. Give details of rotation counter encoding and phase bit encoding.

Behavior common to Leaders and non-Leaders:

If either RC or PB is corrupt in an incoming Token/Frame, the Token/Frame is stripped. When Tokens are created, the Rotation Counter and Phase Bit are copied from an internal register. When Frames are created, the Rotation Counter is copied, but the Phase Bit is 0.

Non-Leader behavior:

While repeating Tokens/Frames/fragments, non-Leaders copy the Rotation Counter and Phase Bit into an internal register (if they are not corrupt). If this changes the register, and the internal phase bit is now 1, then the message carries a token, which the station may optionally capture by changing the Phase Bit of the message to a 0 before repeating it. (A full Frame never carries a token in normal operation--only during stabilization, so it is not necessary to repeat the rest of the message after the Phase Bit.)

Leader behavior:

While repeating Tokens/Frames/fragments, the Leader copies in the other direction--from the internal register into the Rotation Counter and Phase Bit of the message. It also compares the original values in the message to the ones in its internal register. If they are equal, the Leader increments its internal rotation counter, and the message carries a token, which the Leader automatically captures (but which can be released effortlessly). Note that the Leader's internal phase bit is always 1. When the Leader's transmitter is idle, it repeatedly sends Tokens rather than just Idle symbols (which is why releasing an unuseful token is effortless).

4.4.3 Valid-Transmission Timer (TVX).

Perhaps remove this section. TVX is not needed, and may not be worth the complexity. It is primarily useful for fast recovery from Leader crashes. TVX is also mention in sections 5.3.2 R(01), 5.3.5 R(34), 5.3.7 R(51c), 5.4.1 T(04), 5.4.2 T(14), 5.4.3 T(24), 5.4.4 T(34).

5.1.3 Frame Stripping.

Change "PA, SD, FC, DA, and SA" to "PA, SD, FC, RC, PB, DA, and SA". Stations act on RC and PB fields in fragments just as they do in complete Frames.

5.1.5.1 Claim Token Process.

The winner of the bidding is the Leader. The Leader sets its internal phase bit to 1.

5.3.1 Token and Frame Validity Criteria

(1) Token
  (a) unchanged
  (b) unchanged
  (c) Has 6 additional Data Symbols forming valid RC and PB fields.
  (d) remove
(2) Frame (included in Frame_Ct)
  (a) unchanged
  (b) unchanged
  (c) Has valid RC and PB fields.
  (d) same as old (c)
  (e) same as old (d)
(3) unchanged
(4) unchanged
(5) Valid Data Length
  (a) Is an integral number of Data Symbol pairs (octets) between Starting and Ending Delimiters (or after Starting Delimiter if there is no Ending Delimiter).
  (b) Is a minimum number of octets as follows:

FC	Interpretation	Minimum Octets
CLFF ZZZZ to ZZZZ		
0X00 0000	Void	4
1X00 0000	Token	4
0000 0001 to 1111	SMT frame	12
0100 0001 to 1111	SMT frame	20
1000 0001 to 1111	MAC frame	16
1100 0001 to 1111	MAC frame	24
X001 XXXX	LLC frame	12
X101 XXXX	LLC frame	20
X010 XXXX	Implementor frame	12
X110 XXXX	Implementor frame	20
X011 XXXX	Reserved	12
X111 XXXX	Reserved	20

(6) unchanged
(7) unchanged
(8) unchanged

5.3.4 State R2: RC_FR_CTRL (Receive Frame Control Field)

Unchanged until...

R(21b): Format Error.

This transition ... occurs if [unchanged] or if an error is detected in the Rotation Counter or Phase Bit fields.

R(23): Start of Frame

Rename this transition to Start of Frame or Token.

Non-Leaders:

If no errors are detected in the incoming Rotation Counter and Phase Bit, copy them into an internal register, noticing whether the register changes. If it does, and the internal phase bit is now 1, a token has been received, so take the actions of old R(51c).

Leader:

If no errors are detected in the incoming Rotation Counter and Phase Bit, compare them to internal register (whose phase bit is always 1). If they are equal, a token has been received, so increment the internal rotation counter and take the actions of old R(51c).

If the FC field indicates a next station addressing frame, then the N_Flag shall be set.

R(25) Start of Token

Remove this transition.

5.3.5 State R3: RC_FR_BODY (Receive Frame Body).

Unchanged until...

R(31b): Format Error.

This transition ... occurs if [unchanged] or if the maximum PDU length of 9000 symbols is exceeded.

5.3.6 State R4: RC_FR_STATUS (Receive Frame Status).

Add the transition:

R(41g): Format Error.

This transition to state R1 occurs if the maximum PDU length of 9000 symbols is exceeded. On this transition, the FO_Error signal shall be generated and the Lost_Ct shall be incremented.

5.3.7 State R5: CHECK_TK (Check Token).

Remove this section.

5.4.1 State T0: TX_IDLE (Transmitter Idle).

The only thing that changes in this state is that the Leader transmits tokens instead of just idle symbols. This requires the Leader to have a buffer between input and output big enough to hold a Token (27 symbols including a 16-symbol preamble), and to enqueue an entire Token in one atomic action whenever the buffer is empty. But if a token has just been received (TK_Received signal), the Leader should instead merely enqueue an idle symbol (whether the buffer is empty or not). This is crucial if the token is usable (lest the Leader immediately give it away), and sensible even if the token is unusable (since the transition to state T3 ISSUE_TK will cause a token to be sent anyway). Likewise, when a FR_Strip signal is received, the Leader should enqueue an idle symbol to ensure that the fragment boundary is still marked for the next station.

5.4.2 State T1: REPEAT (Repeat).

Introduction is unchanged except for the third sentence:

If a usable token is detected, then the transmitter of a non-Leader changes the Phase Bit to 0 before repeating it and thus acquires the token. The Leader always overwrites the Rotation Counter and Phase Bit from its internal register.

T(10a): Token Capture.

Renumber this section to T(12).

A transition to State T2 occurs if a token has been received (TK_Received signal) if the token is usable. A token is usable when the ring is operational (Ring_Operational) and either:

(1) unchanged
(2) unchanged
(3) unchanged

At this point, the station captures the token by changing the Phase Bit to 0 before it is repeated (except that the Leader always uses its internal phase bit value, which is 1). After the Phase Bit is transmitted, the actions of T(02) are performed, followed by the change to state T2.

Other transitions are unchanged.

5.4.3 State T2: TX_DATA (Transmit Data).

Introduction unchanged except for this addition:

Use the internal register to generate the Rotation Counter for the new Frame. The Phase Bit shall be 0.

Caveat: It is quite possible that the receiver will try to write to the register at the same time that the transmitter is trying to read from it. The reads and writes need to be atomic to avoid sending corrupt values. Scratch space may be necessary.

T(23): Done.

....This transition shall also be taken if [unchanged], or on a TK_Received signal, or if the internal phase bit is 0.

5.4.4 State T3: ISSUE_TK (Issue Token).

Introduction is unchanged except for this addition:

Use the internal register to generate the Rotation Counter and Phase Bit for the new Token. (The internal phase bit should normally be 1 in this state, but might be 0 if we are stabilizing.)

Transitions are unchanged except for this one:

T(30b): Token Issued.

A transition to State T0 occurs after the transmitter has issued the Phase Bit of the Token.

5.4.5 State T4: CLAIM_TK (Claim Token).

Upon entering this state, the Leader_Flag is cleared, [unchanged].

T(43) Successful Claim.

Insert: Leader_Flag and Phase Bit shall be set.

Appendix B

Correctness of the One-to-Many Sliding Window

The stabilization of the one-to-many sliding window protocol is very similar to the stabilization of the two-node sliding window protocol. The following proof of correctness refers to the actions of Figure 4.8, and the concepts of "earliest" and "latest" introduced in Subsection 4.4.3. All arithmetic is modulo M, where $M > 2w$.

B.1 Definitions

The terms *dataparent* & *datachild* and *ackparent* & *ackchild* refer to neighbors of a node in the data and ack trees, respectively. A receiver's *data path* is the path through the data tree from the sender down to the receiver, including all queues Q_j along the way. A receiver's *ack path* is the path through the ack tree from the receiver up to the sender, including all stored acks A_j along the way.

Assume that acks (whether in transit or stored in A_j variables) are represented in such a way that each ack must contain exactly one $\langle i, ack\rangle$ pair for each node i below it in the ack tree, and no other pairs. The digest representation used in Subsection 4.4.3 neatly satisfies this requirement by causing the i values to disappear altogether.

In a given state:

- The *upward sequence* of receiver i is the sequence of values starting with R_i and proceeding through all the *ack* values paired with i in all the in-transit acks and A_j values on i's ack path.
- The *downward sequence* of receiver i is the sequence of values starting with L and proceeding through all the L' values in the in-transit and queued data packets on i's data path.
- The *validity sequence* of a receiver is the concatenation of its upward and downward sequences, in that order.

A sequence of bounded integers is *non-increasing* iff each successive value is either equal to the previous value, or does not occur in the range $[p, f]$, where p is the previous value and f is the first value. For example, $\langle 4, 3, 3, 0, 99, 43, 6\rangle$ is non-increasing, and so is $\langle 4, 6\rangle$, but $\langle 4, 5, 6\rangle$ is not, because 6 *is* in $[5, 4]$ in modular arithmetic.

A receiver is *valid* in a given state iff it meets the following conditions:

1. For every data packet in transit or queued on the receiver's data path, $seq \in [L' + 1, L' + w]$.
2. The receiver's validity sequence is non-increasing.
3. The receiver's validity sequence spans a range of at most w, which is to say, every value in the sequence is in $[R_i - w, R_i]$. Equivalently, every value in the sequence is in $[L^*, L^* + w]$, where L^* is the last value in the sequence.

A *valid state* is one in which every receiver is valid.

B.2 Safety and Liveness

Conditions 2 and 3 together imply that every value in the upward sequence is in $[L, L + w]$. Therefore, the sender ack check has no effect on a valid state.

Condition 3 implies that $R_i \in [L', L' + w]$ for any data packet arriving at receiver i. Therefore, the receiver window adjustment has no effect on a valid state.

Any of the six of the actions of the protocol, applied to a state in which receiver k is valid, produces a state in which receiver k is valid, as shown here:

Condition 1 is obviously preserved by $\mathsf{SendData}_{0j}$, and unaffected by all other actions.

$\mathsf{SendData}_{0j}$ preserves conditions 2 and 3 because it merely duplicates a value in the validity sequence.

$\mathsf{SendData}_{ij}$ preserves conditions 2 and 3 because it doesn't change the validity sequence.

$\mathsf{SendAck}_i$ preserves conditions 2 and 3 because, if it affects the validity sequence at all (i.e, if receiver i is in the ancestry of or equal to receiver k), it only copies a value.

$\mathsf{ReceiveAck}_{ji}$ preserves conditions 2 and 3 because, if it affects the validity sequence at all, it only deletes a value.

$\mathsf{ReceiveAck}_{j0}$ preserves conditions 2 and 3 because it first deletes a value in the validity sequence, then possibly changes one value in the sequence, but in a safe way. In order to violate condition 2, there would have to be a pair $\langle h, ack \rangle$ in one of the sender's A_j variables such that the new L is not ack but is in $[ack, R_h]$. According to the code, after L changes, every ack value in the sender's A_j variables must be in $[L, L+w]$. We also know that that ack is in $[R_h - w, R_h]$. Together, these range restrictions imply that $\langle L, L+w, ack, R_h - w, R_h \rangle$ is non-increasing, which cannot be true because the modulus is greater than $2w$. In order for $\mathsf{ReceiveAck}_{j0}$ to violate condition 3, L would have to be the last value in the validity sequence, but even then, the range of the sequence can only shrink, not grow.

$\mathsf{ReceiveData}_{ij}$ doesn't change any of the validity sequences that continue through the queues, and merely deletes a value from the sequence that ends at node i. It can also affect the first value in the sequence beginning at node i. The assumption of an initial valid state implies that the receiver window adjustment never applies; R_i changes only when $R_i + 1 = seq$, in which case R_i increments. This could violate condition 2 only if the sequence then wraps completely around the bounded integer space. But using condition 1, we see that condition 3 is preserved, so the sequence could not possibly wrap around. $\Box$

Furthermore, from the definition of a valid state, it follows that any of the six actions, applied to a valid state, produce a valid state.

Valid states may be stable, but do they deserve to be called "valid"? Yes. Notice that starting from any valid state, data packets are accepted in order, because they are accepted only when $seq = R_i + 1$, which causes R_i to increment. Also note that liveness is guaranteed: if L tries to get stuck, the sender will eventually send a data packet with $L' = L$ and $seq = L + 1$ to each of its datachildren, which will cause all the R_i variables to enter the range $[L+1, L+w]$ if they were not already in that range. R_i cannot leave that range as long as L is stuck. Every receiver will eventually send an ack in that range, causing all ack values in all A_j variables to enter that range, which will cause L to advance. Thus, L can remain stuck for at most one round-trip delay[1], plus the time to transmit a window of data.

The claim of liveness above, and the time bounds on self-stabilization that will be derived below, both depend on data always being available to send. If this is not actually the case, but we still want time bounds, we can allow the sender, whenever the its message queue becomes empty, to enqueue a null message that does not affect the data stream. Artificial delays can accompany such null messages so that they do not consume too much bandwidth (a null message must still carry L' and seq). Alternately, we may not care about liveness and stabilization while there is no data to send.

B.3 Stabilization

We need to show that an arbitrary state will reach a valid state. Let us focus on an arbitrary receiver i, and show that it will become valid.

Call the initial state $\mathcal{S}_0$.

Consider the first data packet sent. Wait until it arrives at receiver i, producing state $\mathcal{S}_1$. Note that condition 1 is true in $\mathcal{S}_1$, and that the code for $\mathsf{SendData}_{0j}$ insures that it will stay true thereafter. After $\mathcal{S}_1$, wait until receiver i sends an ack and it is received by ackparent(i), producing state $\mathcal{S}_{1.1}$. Then wait until ackparent(i) sends an ack and it is received by ackparent(ackparent(i)), producing state $\mathcal{S}_{1.2}$. Continue in this manner until it is the root which is receiving the ack, producing state $\mathcal{S}_{1.d}$, which we will call state $\mathcal{S}_2$.

[1] The round-trip delay of the system is the greatest round-trip delay between the sender and each receiver.

Let L_0 and L_2 be the sender's L values in states $\mathcal{S}_0$ and $\mathcal{S}_2$.

How many distinct *ack* values could the sender have received during this time? Notice that receiver i can change R_i only when it receives a packet. The answer, therefore, is no more than one for each ack in i's ack path in state $\mathcal{S}_0$, plus one for each A_j value in i's ack path in state $\mathcal{S}_0$, plus one for the R_i value in state $\mathcal{S}_0$, plus one for each data packet in i's data path in state $\mathcal{S}_0$ (including queued packets), plus one for the ack whose arrival produced state $\mathcal{S}_2$. Assuming that the links are bounded, call the maximum possible value of this sum $c_{max} - 1$.

Let us assume that the modulus M is larger than $w \cdot c_{max}$.

Because of the $\mathsf{ReceiveAck}_{0j}$ code, the sender can advance L by at most w for each distinct *ack* value associated with i. Hence between states $\mathcal{S}_0$ and $\mathcal{S}_2$, L could have advanced by at most $w(c_{max} - 1)$. Note that ever since state $\mathcal{S}_1$, every L' value in i's data path has been in $[L_0, L_2]$, and every *seq* value has been in $[L_0 + 1, L_2 + w]$. Because receiver i, upon receiving a packet, must set R_i to *seq* unless R_i was in the range $[L', L' + w]$, we know that R_i has been in the range $[L_0, L_2 + w]$ ever since state $\mathcal{S}_1$, which means that in state $\mathcal{S}_2$, all *ack* values associated with i in i's ack path are in $[L_0, L_2 + w]$. Thus, in state $\mathcal{S}_2$, every value in i's validity sequence is in the range $[L_0, L_2 + w]$.

Because the modulus M is larger than $w \cdot c_{max}$, we can define a total ordering on the values in the range $[L_0, L_2 + w]$, with L_0 as the "zero" value, and each successive value being the next larger value.

We can now use words like "increase" and "decrease" to describe variables than stay in this range. For instance, during the time interval in question, from state $\mathcal{S}_0$ to state $\mathcal{S}_2$, L never decreases. Hence i's downward sequence in state $\mathcal{S}_1$ is non-increasing. In state $\mathcal{S}_1$, receiver i has just received a data packet containing some L' value, which means that R_i must be in the range $[L', L' + w]$ (whether it accepted the packet or not—see the $\mathsf{ReceiveData}_{ij}$ code). Because each successive packet that it receives until state $\mathcal{S}_2$ contains an L' value larger than (or equal to) the previous one, R_i is always $\leq L^* + w$, where L^* is the last value in i's downward sequence. Therefore, subsequent changes to R_i can only be increases, never decreases. Consequently, in state $\mathcal{S}_2$, i's upward sequence is non-increasing.

$$time(\mathcal{S}_2) - time(\mathcal{S}_0) \leq \textit{round-trip delay}$$

What happens after $\mathcal{S}_2$? L will remain stuck at L_2 until the *ack* value associated with i in the sender's A_j variables enters the range $[L_2+1, L_2+w]$. Until that happens, we preserve the property that all values in i's validity sequence are in the range $[L_0, L_2+w]$, and the non-increasing property of i's downward and upward sequences, and therefore we preserve the property that $R_i \leq L^* + w$. Denote by $\mathcal{S}_3$ the first state after $\mathcal{S}_2$ (or equal to $\mathcal{S}_2$) in which the *ack* value associated with i at the sender is in the range $[L_2, L_2+w]$. All the properties just mentioned will have been preserved through state $\mathcal{S}_3$ (L may have increased from L_2, but it is still no more than L_2+w). We also know that we must reach $\mathcal{S}_3$. Why? Suppose we never do. Then L will remain at L_2 forever. A data packet with $L' = L_2$ will make its way from the sender to receiver i, causing R_i to enter the range $[L_2, L_2+w]$. R_i never decreases, and stays in the range $[L_0, L_2+w]$, so it must stay in the range $[L_2, L_2+w]$. As acks propagate up i's ack path, all the *ack* values associated with i will enter this range, including the one in the sender. At this point, we have reached $\mathcal{S}_3$.

$$time(\mathcal{S}_3) - time(\mathcal{S}_2) \leq \textit{round-trip delay}$$

Examine $\mathcal{S}_3$. Receiver i's downward and upward sequences are non-increasing. The last value in the upward sequence (the *ack* value associated with i at the sender) is $\geq L$, so condition 2 is satisfied. $R_i \leq L^* + w$, so condition 3 is satisfied. Condition 1 has been satisfied since state $\mathcal{S}_1$. Receiver i is therefore valid in state $\mathcal{S}_3$. It has taken at most two round-trip delays to get there from state $\mathcal{S}_0$.

We have previously shown that once a receiver becomes valid, it stays valid. The stabilization proof applies independently to each receiver. Therefore, within two round-trip delays, the system is in a valid state.

References

[1] Y. Afek, S. Kutten, and M. Yung. Memory-efficient self-stabilization on general networks. In *Proc. 4th Workshop on Distributed Algorithms*, pages 15–28, Italy, September 1990. Springer-Verlag (LNCS 486).

[2] American National Standards Institute. *Fiber Distributed Data Interface (FDDI) Token Ring Media Access Control (MAC)*, November 1987. ANSI X3.139-1987.

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

[4] E. W. Dijkstra. Self stabilization in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.

[5] S. Floyd, V. Jacobson, S. McCanne, C. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proc. ACM SIGCOMM'94 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 342–356, Cambridge, August 1995.

[6] M. G. Gouda and N. J. Multari. Stabilizing communication protocols. Technical Report TR-90-20, Dept. of Computer Science, Univ. of Texas at Austin, June 1990.

[7] Raj Jain. *FDDI Handbook: High-speed Networking Using Fiber and Other Media.* Addison-Wesley Publishing Company, Reading, MA, 1994.

[8] S. Katz and K. Perry. Self-stabilizing extensions for message-passing systems. In *Proc. 10th ACM Symp. on Principles of Distributed Computing*, Quebec City, August 1990.

[9] H. T. Kung, T. Blackwell, and A. Chapman. Credit-based flow control for ATM networks: Credit update protocol, adaptive credit allocation, and statistical multiplexing. In *Proc. ACM SIGCOMM'94 Conf. on Communications Architectures, Protocols, and Applications*, pages 101–114, London, August 1994.

[10] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

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

[12] A.N. Netravali, W.D. Roome, and K. Sabnani. Design and implementation of a high speed transport protocol. In *IEEE Transactions on Communications*, volume 38, pages 2010–2024, November 1990.

[13] R. Perlman. An algorithm for distributed computation of a spanning tree in an extended LAN. In *Proc. 9th Data Communication Symposium*, pages 44–53, September 1985.

[14] Radia Perlman. Fault tolerant broadcast of routing information. *Computer Networks*, December 1983.

[15] Radia Perlman. *Network Layer Protocols With Byzantine Robustness*. PhD thesis, MIT, Laboratory for Computer Science, August 1988.

[16] E. C. Rosen. Vulnerabilities of network control protocols: An example. *Computer Communications Review*, July 1981.

[17] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25, March 1993.

[18] J. M. Spinelli. Self-stabilizing ARQ on channels with bounded memory or bounded delay. *Proc. of IEEE INFOCOM'93*, 1993.

[19] J. M. Spinelli. Self-stabilizing sliding window ARQ protocols. *Submitted for publication*, 1994.

[20] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, N. J., 1981.

[21] G. Varghese. *Self-stabilization by Local Checking and Correction*. PhD thesis, Massachusetts Institute of Technology, October 1993.

[22] G. Varghese. Self-stabilization by counter flushing. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, August 1994.

Vita

Adam M. Costello

Date of Birth 18 April 1971

Place of Birth Pensacola, Florida

Degrees B.S. Computer Science, May 1993
B.S. Electrical Engineering, May 1993

Publications Costello, A. M. and G. Varghese. Self-Stabilization by Window Washing. In *Proc. 15th ACM Symp. on Principles of Distributed Computing*, Philadelphia, May 1996.

Costello, A. M. and G. Varghese. Redesigning the BSD Callout and Timer Facilities. Technical report WUCS-95-23, November 1995.

Boutell, T. *et. al.* PNG (Portable Network Graphics) Specification. Version 1.0. W3C Proposed Recommendation 01-July-1996.

Web Page `http://www.cs.berkeley.edu/~amc/`

August 1996