# Self-Stabilization by Window Washing

Adam M. Costello
<amc@cs.wustl.edu>
http://www.cs.wustl.edu/~amc/

George Varghese
<varghese@cs.wustl.edu>
http://www.ccrc.wustl.edu/~varghese/

Washington University in St. Louis
http://www.wustl.edu/

## Abstract

A useful way to design simple and robust protocols is to make them self-stabilizing. We describe a new general technique for self-stabilization called *window washing*. We apply this technique to generalized sliding window protocols that work on a number of topologies. This results in simple, efficient, and self-stabilizing protocols. As far as we know, both window washing and generalized sliding window protocols are new ideas. Our protocols can be used for data links, reliable broadcast, and flow control.

## 1 Introduction

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 [Ros81, Per83]. 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.

In this paper we describe a self-stabilizing version of the well-known sliding window protocol. We also describe a new *pipelined* reliable broadcast protocol, by generalizing the sliding window protocol to work with multiple receivers. To the best of our knowledge, even the non-stabilizing version of our one-to-many sliding window protocol is new. We also show how small modifications achieve self-stabilization in the Credit Update Protocol (CUP) for flow control [KBC94], which has been implemented in Bell-Northern Research ATM switches. Underlying these practical applications is a new theoretical technique that we call *window washing*.



We had earlier [Var94] developed a technique called *counter flushing* which allowed a single leader to broadcast *one* message at a time to the receivers. Such stop-and-wait protocols are inefficient in networks where the propagation delay is large compared to the transmission delay. Our new window washing technique allows the sender to pipeline a *window* of packets, improving throughput. In its simplest form with only one receiver, this is just a sliding window protocol [Tan81]. We consider the natural generalization: a one-to-many sliding window protocol with multiple receivers, and the application of our technique to make such a protocol self-stabilizing. One-to-many sliding window protocols would be useful for reliable broadcast, for example in ATM or IP networks.

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.

**Previous Work** Self-stabilizing protocols were introduced by Dijkstra [Dij74, Sch93]. There have been few general techniques for self-stabilization. Katz and Perry [KP90] showed how to compile an arbitrary asynchronous protocol into a stabilizing equivalent. Their general transformation is expensive; hence more efficient and less general techniques are useful. Techniques that transform any *locally checkable* protocol into a stabilizing equivalent are given in [AKY90, APV91, Var93]. However, our work on WW applies to protocols that are not locally checkable.

Gouda and Multari [GM90] describe a two-node sliding protocol using *unbounded* integers. Real protocols require bounded integers. Spinelli [Spi93] describes a pair of two-node self-stabilizing sliding window protocols with bounded counters. Our sliding protocol stabilizes using a completely different mechanism than Spinelli's. More interestingly, our stabilizing mechanism is a general idea 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 the CUP flow control protocol [KBC94]. Our protocols stabilize faster, taking at most two *actual* round-trip delays to stabilize, whereas Spinelli's may take up to two *worst-case* round-trip delays.[1] In some real links the difference between a typical round-trip delay and a worst-case round-trip delay can be 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.

The rest of the paper is organized as follows. Section 2 describes our model. Section 3 reviews counter flushing. Section 4 shows how window washing works for a two-node sliding window protocol. Section 5 describes the one-to-many stabilizing sliding window protocol. Section 6 shows how CUP, with small modifications, uses window washing to achieve self-stabilization. Section 7 states our conclusions. Appendix A details a proof of the two-node stabilizing sliding window protocol. Appendix B presents an optimization for avoiding an implosion of acks in the one-to-many case.

# 2 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., [Per85] calculates a leader in $O(D)$ time.) We model the nodes and links of the network using Input/Output Automata (IOA) [LT89].

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 from node $i$ toward node $j$, and an input action $\mathsf{Receive}_{i,j}(p)$ to receive a packet[2] 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; 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.

[1] The $st/rst$ mechanism may take up to two worst-case round-trip delays to detect a bad state. In the go-back-$n$ version of Spinelli's protocol, *RESET*s are not possible; in the selective repeat version, they add a third worst-case round-trip delay plus an actual round-trip delay to the stabilization time.

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

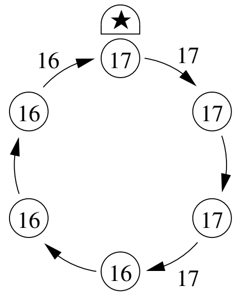


Figure 1: A valid state of a token ring using counter flushing.

# 3 Counter Flushing

As a point of departure, we review counter flushing [Dij74, Var94]. The abstract ideas are:

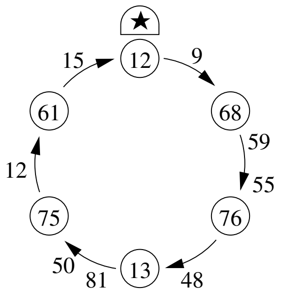
- Nodes periodically inform their neighbors of "where" in the execution they are.
- The leader node is always "right" by definition.[3]
- Whenever a discrepancy is discovered, the leader ignores it, but a non-leader will re-sync by altering its own state.

The details 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 solid band, starting with the leader and continuing part-way around the ring, contains only the value $x$, while the rest of the ring contains only the value $x - 1$. The token is at the end of the first band. For example, Figure 1 shows a valid state with the token in transit from node 2 to node 3.

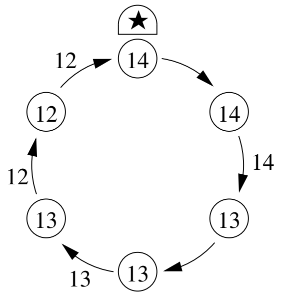
When a non-leader node $i$ receives a token packet from upstream, it compares the counter value in the packet ($c$) with

[3] Recall that our model assumes that there is a distinguished network node that is the leader.

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):

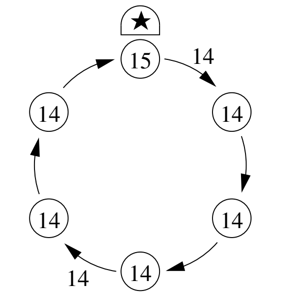



Figure 2: Stabilization of the token ring using counter flushing.

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 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.

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 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 solid band 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 shows an example stabilization scenario.

Note that a token passing protocol can be used to broadcast a data message (carried in the token packet) from the leader to the other nodes. (In the case of a two-node ring, this reduces to the famous stop-and-wait data link protocol [Tan81].) However, a message must be delivered to all receivers before the next message can be sent. Thus the early work on counter flushing [Var94] applied to the problem of broadcasting a *single* message at a time to a set of receivers. We now show how to generalize this approach to allow *pipelining*.

# 4 Window Washing

A sliding window protocol [Tan81] 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 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.

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 [Tan81]. 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 $w$ [Tan81]. However, for self-stabilization we will require a larger value. The code for this protocol is shown in Figure 3. We have not modeled the arrival of messages from users 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 some-

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 $m$
endif

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

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

Figure 3: Sliding window code.

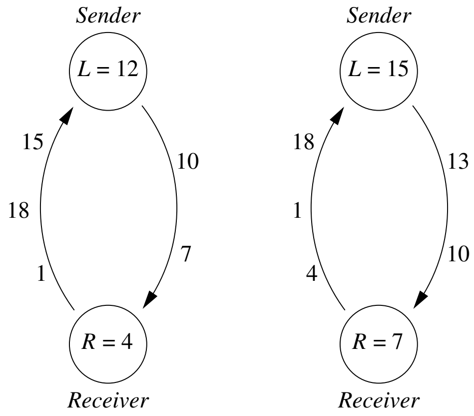



Figure 4: Livelock caused by a too-small modulus. The numbers on the downward links are $seq$ values. One execution of each action transforms the state on the left into the state on the right, which is identical except that every value has increased by the window size 3.

times send an ack whenever a new packet is accepted; our protocol can simulate such behaviors. Although in our protocol, it appears that the 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.

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 to 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 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*.

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, 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

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 $m$
endif

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

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

Figure 5: Stabilizing sliding window code. Changes from the original code are boxed.

$M \geq w \cdot c_{max}$ we foil the scenario described. However, the requirement $M > w \cdot c_{max}$ will make the proof easier.

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 5. 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. 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). A proof of stabilization is provided in Appendix A.

# 5 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 broadcast values to many receivers. However, most current protocols[4] *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 could potentially be implemented in ATM switches and IP routers.

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[5] 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 $i$ towards the sender. Figure 6 shows the data and ack trees for a ring topology. If the physical topology were a tree (with the sender at the root), the data tree would contain all the leafward links, and the ack tree would contain all the rootward links. Arbitrary physical topologies are permissible.

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 7.

[4] A few recent Internet protocols like SRM [FJM+95] offer reliable multicast, but these protocols have not been proven to be self-stabilizing.

[5] We can generalize these notions to acyclic graphs, but that would clutter

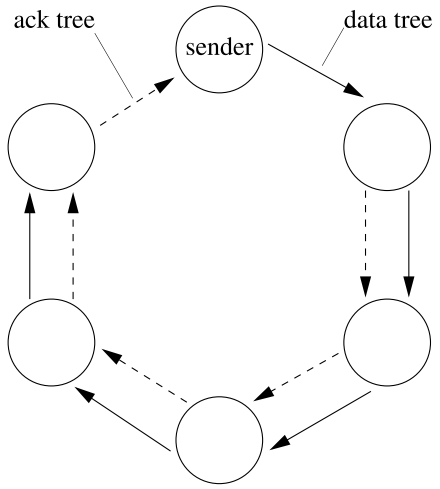



Figure 6: The data and ack trees for a clockwise ring topology.

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. However, it is possible to package up a an arbitrarily large set of $ack$ values into a fixed-size *digest* which captures just the information needed by the protocol. This leads to $O(1)$ space and $O(1)$ bandwidth, thus avoiding the implosion problem. The details of this optimization are presented in Appendix B.

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 for the one-to-many protocol is not included here, because it is so similar to the proof for the two-node protocol.

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 $m$
endif
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 $Q_j$

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

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

$\mathsf{ReceiveAck}_{0j}(ack)$ (* Sender absorbs ack from ackchild $j$. *)
Effects:
$A_j \leftarrow ack$
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
endif

Figure 7: Stabilizing one-to-many sliding window code. A *datachild* is a child in the data tree, an *ackparent* is a parent in the ack tree, etc.

# 6 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 of $R$, 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** 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.

the presentation.

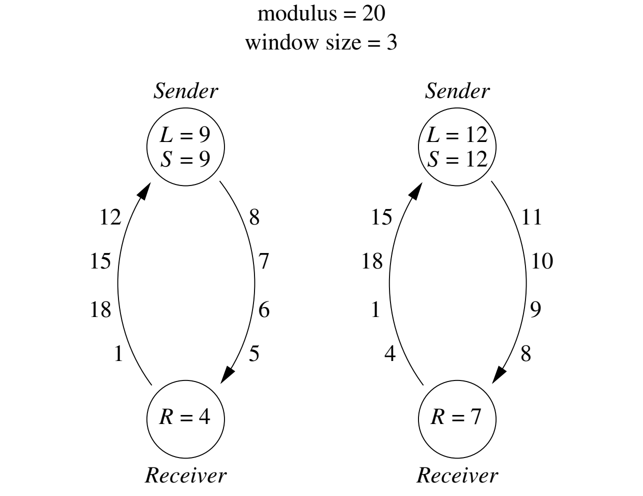



Figure 8: Livelock caused by a too-small modulus. $w$ executions of SendData and ReceiveData plus one execution of SendAck and ReceiveAck transform the state on the left into the state on the right, which is identical except that every value has increased by the window size 3.

**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 is shown in Figure 8. It is foiled if $M > w \cdot r_{max} + f_{max}$, where $r_{max}$ is the maximum number of acks that fit in the reverse link plus one, and $f_{max}$ is the maximum number of data packets that fit in the forward link plus one.[6]

[6] 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.

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

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

SendSync$(seq)$ (* Sender emits sync packet. *)
  Preconditions:
    $seq = S$

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

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

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

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

Figure 9: Stabilizing sliding window code for flow-control without reliability.

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 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.

The simplified code is shown in Figure 9. It is not new—it is virtually identical to the Credit Update Protocol (CUP) [KBC94] 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 9: the latter assumes a cheap $length(Q)$ function. CUP never counts the occupancy of the queue (presumably because that 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.

# 7 Conclusions

There are two main contributions of this paper. First, while a bounded integer sliding window protocol was described by Spinelli, our protocol is simpler, faster, and based on a more general mechanism than that of Spinelli. Second, we have generalized sliding window protocols to the multiple receiver case and shown that our stabilizing techniques apply to the general case. Additionally, we have provided evidence that the window washing mechanism is versatile, by applying it to the problem of flow control.

We have argued that the stabilizing protocols produced by our techniques are useful for real networks. Our sliding window protocol requires only minor changes to the usual implementation in order to make it stabilizing. The reliable broadcast protocol seems simple enough to implement even on ATM switches; recall that only the sender does retransmission, so switches do not have to buffer packets. Our goal is to bridge the gap between elegant theoretical techniques for fault-tolerance and the needs of real networks.

# A Stabilization Proof for the Two-Node Sliding Window

This section proves that the protocol of Figure 5 stabilizes, and refers to the actions defined there. All arithmetic is modulo $M$, and $M > w$.

## A.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 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.

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

1. For all data packets 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.

## A.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 the 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 there always being data to send. If this is not actually the case, but we still want the 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. Alternately, we may not care about liveness and stabilization while there is no data to send.

### A.3 Self-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 its $R$ value 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 $\mathcal{S}_0$. Assuming that the links are bounded, call the maximum possible value of this sum $c_{max} - 1$. (That makes $c_{max}$ the maximum number of data packets plus acks plus 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 $w$, $L$ 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 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. So 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. Therefore, in state $\mathcal{S}_2$, the upward sequence is non-increasing.

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

What happens after $\mathcal{S}_2$? $L$ will remain stuck at $L_2$ until an $ack$ value arrives in the range $[L_2 + 1, L_2 + w]$. So 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 $\mathcal{S}_3$.

$time(\mathcal{S}_3) - time(\mathcal{S}_2) \leq$ *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 $\mathcal{S}_1$. So $\mathcal{S}_3$ is a valid state. It has taken at most two round-trip delays to get there from state $\mathcal{S}_0$.

## B Avoiding Implosion

At first glance, the one-to-many stabilizing sliding window protocol described in Section 5 appears to require $O(n)$ space for an ack packet, and therefore $O(n)$ bandwidth, where $n$ is the number of receivers. 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.

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

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

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

Figure 10: Replacement stabilizing one-to-many sliding window code using ack digests.

earliest $ack$ value, and $hi$ is the latest. Otherwise, $f(x) = \mathsf{wide}$. Notice that the terms "earliest" and "latest" make sense only if $MaxWidth$ is no more than half the modulus of the bounded integers, so we require that. $MaxWidth$ 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 $f(x_1), f(x_2), f(x_3), \ldots, f(x_k)$ as follows. If any of the $f(x_i)$ is $\mathsf{wide}$, then $f(x) = \mathsf{wide}$. 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, $f(x) = \mathsf{wide}$.

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 $f(x) \neq \mathsf{wide}$, 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 10.

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

# References


[AKY90] 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).

[APV91] 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.

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

[FJM+95] 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.

[GM90] 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.

[KBC94] 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.

[KP90] 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.

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

[Per83] Radia Perlman. Fault tolerant broadcast of routing information. *Computer Networks*, December 1983.

[Per85] 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.

[Ros81] E. C. Rosen. Vulnerabilities of network control protocols: An example. *Computer Communications Review*, July 1981.

[Sch93] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25, March 1993.

[Spi93] J. M. Spinelli. Self-stabilizing sliding window arq protocols. *Proc. of IEEE Infocomm 93*, 1993. Self-stabilizing ARQ protocols on Channels with Bounded Memory or Bounded Delay.

[Tan81] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, N. J., 1981.

[Var93] G. Varghese. Self-stabilization by local checking and correction. Ph.D. Thesis MIT/LCS/TR-583, Massachusetts Institute of Technology, 1993.

[Var94] G. Varghese. Self-stabilization by counter flushing. In *Proc. 13th ACM Symp. on Principles of Distributed Computing*, Los Angeles, August 1994.