Self-Stabilization by Window Washing

Jay Maisel, "Window Washer, Bogota, Columbia"

by Adam M. Costello
(with George Varghese)

Supported by NSF Grant NCR-9405444
and the IPP Consortium

Self-Stabilization

A protocol is *self-stabilizing* if

- Starting from a valid state, it remains in valid states.
- Starting from *any* state, it reaches a valid state within bounded time.

Self-stabilizing protocols are

- **simple** because they handle all errors with one mechanism, and need no initialization.
- **robust** because they recover from arbitrary faults (except program code corruption).

Outline

- **Counter Flushing:** A technique for self-stabilization, e.g. token passing and stop & wait protocols.

- **Window Washing:** An extension of counter flushing to the sliding window protocol.

- **One-to-Many** extension of window washing.

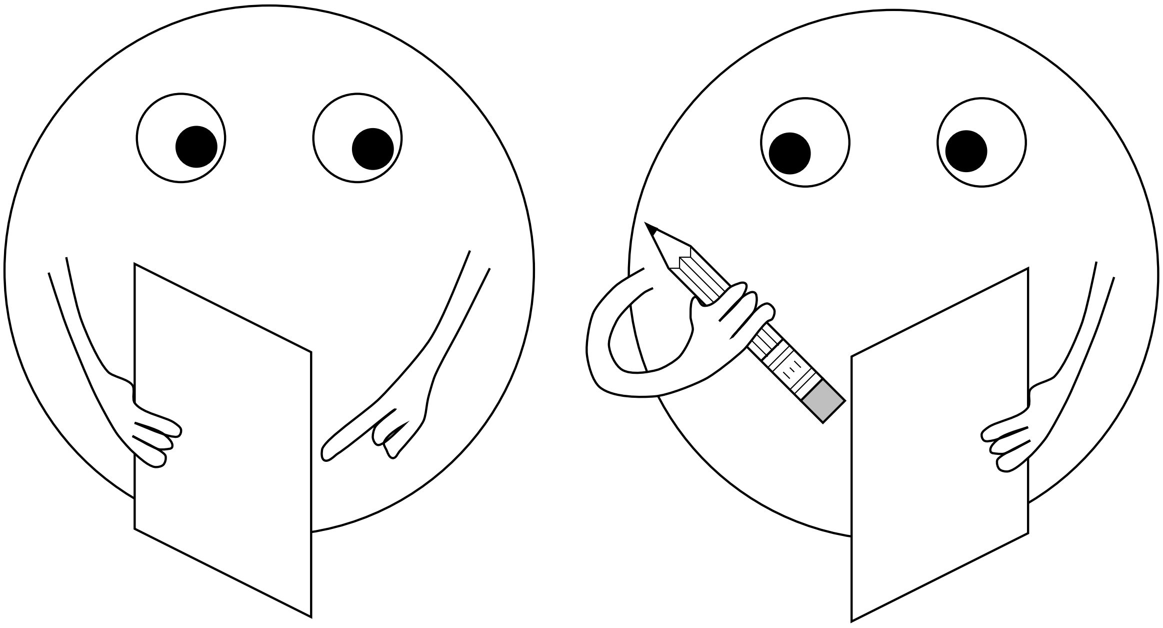
- **Flow Control** application of window washing.

Counter Flushing

One technique to achieve self-stabilization is *counter flushing.*

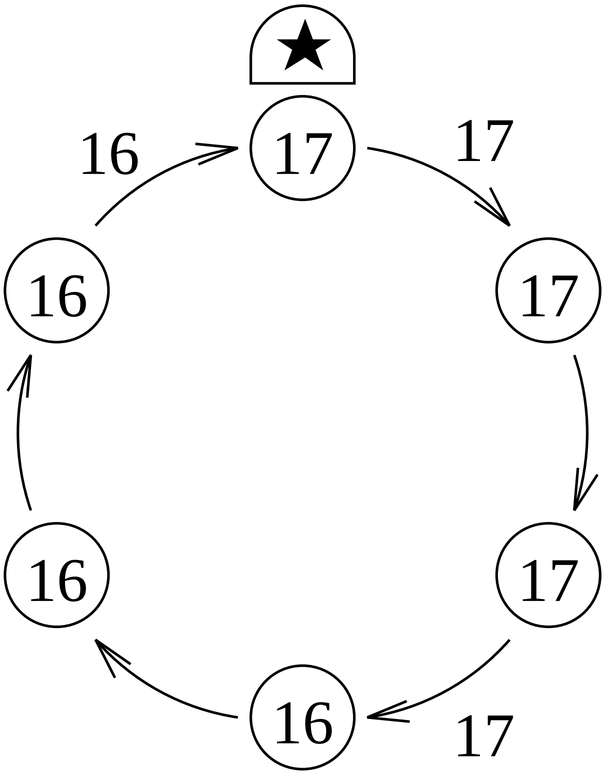
Nodes periodically inform neighbors of "where" in the execution they are.

Nodes who disagree change themselves to re-sync.



There is a "leader" who is always right. It never re-syncs, but it does do sanity checking on incoming messages.

Token Passing with Counter Flushing



"Where in the execution" is the number of times a node has held the token.

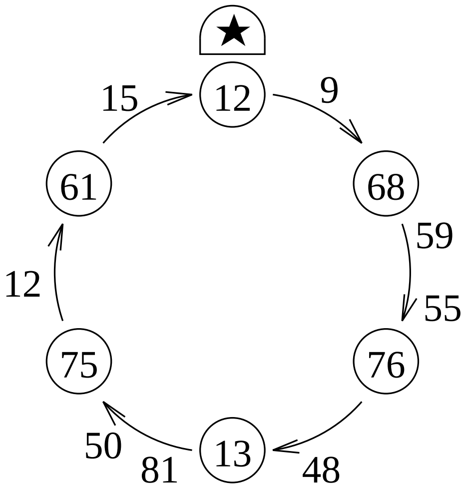
All nodes periodically send their counters.

Non-leaders always copy incoming counters.

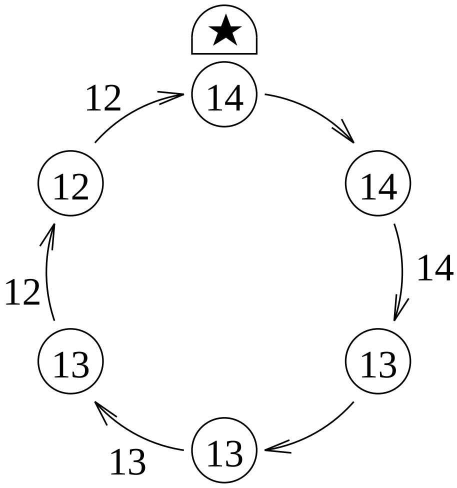
The leader increments when an incoming counter matches its own.

Links must be FIFO, and the modulus must be large enough.

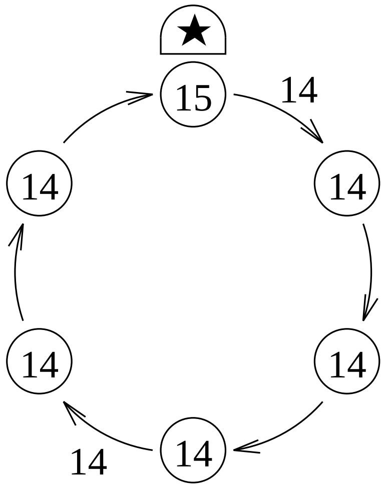
Stabilization of Token Ring



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



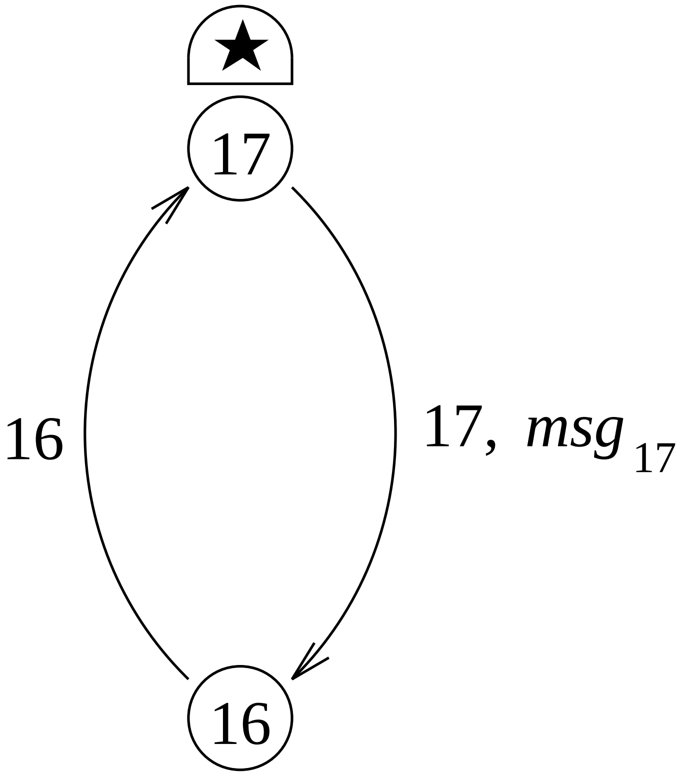
After one round-trip delay, all garbage values have been flushed out.



The leader remains stuck until greatest value fills the ring (at most one more round-trip delay).

From Token Ring to Stop & Wait

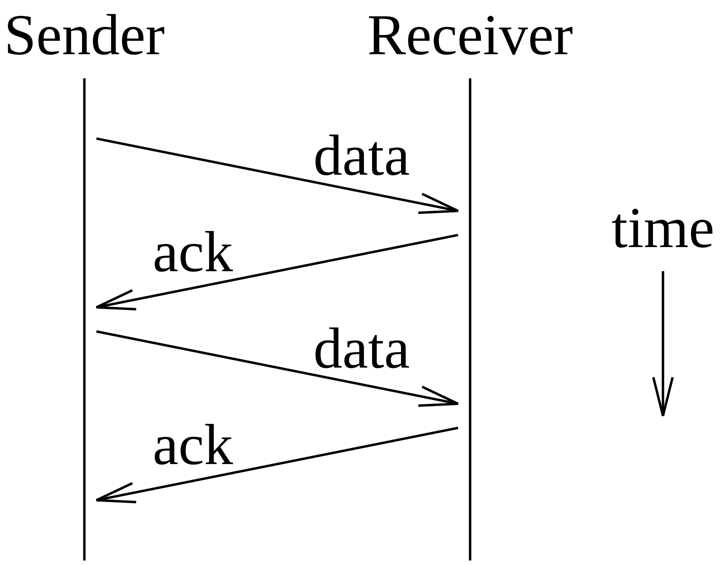
two-node token ring + piggybacked data = stop & wait



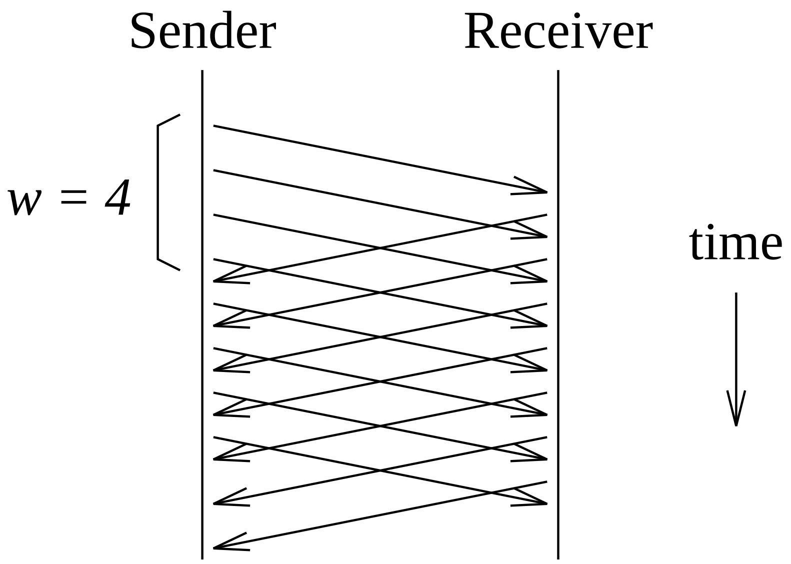
The stop & wait protocol can be extended to the *sliding window* protocol to allow pipelining.

Stop & Wait vs. Sliding Window

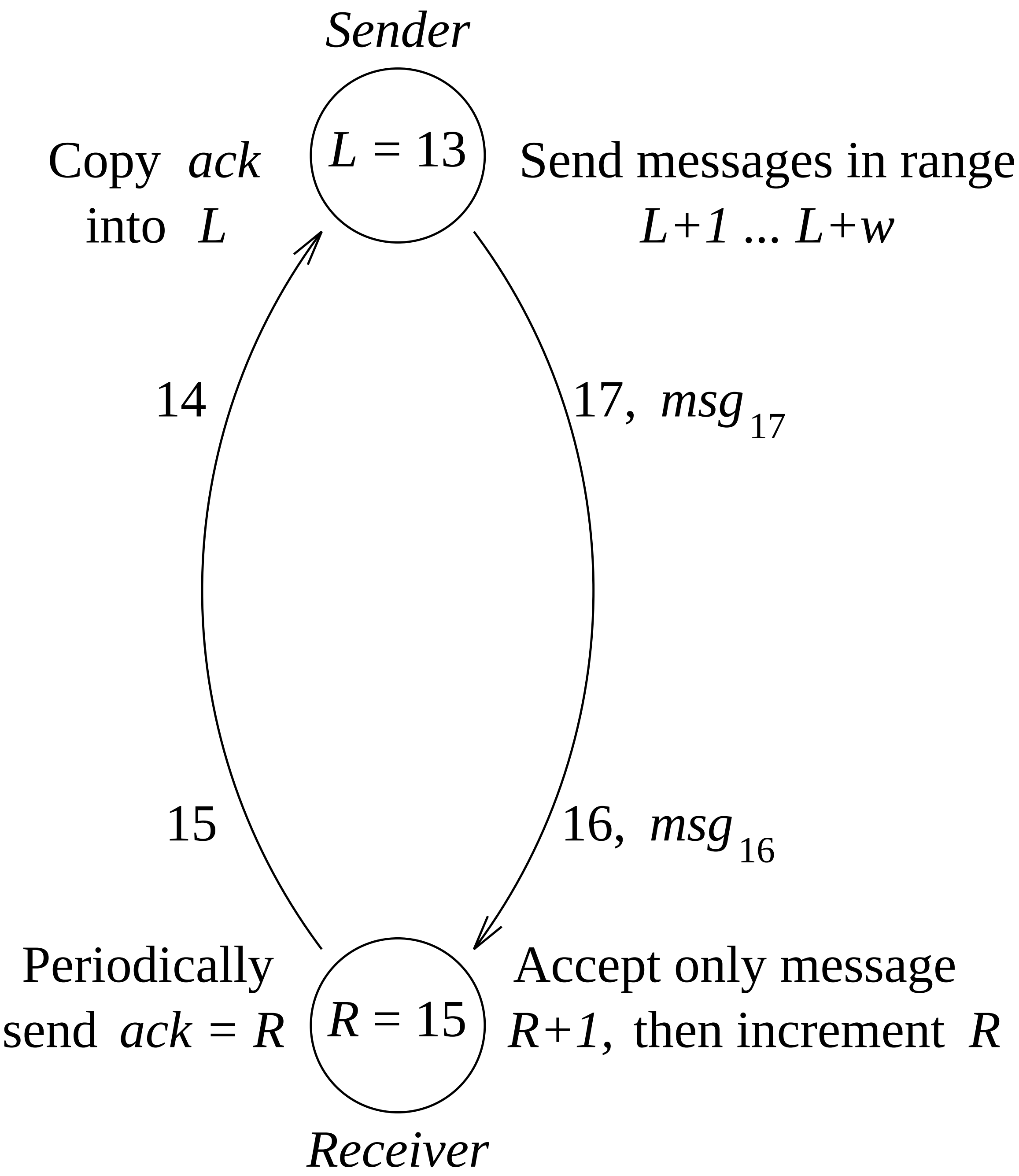
With stop & wait, each message requires one round-trip delay.



With sliding window, each round-trip delay is shared by w messages.



Sliding Window Protocol



Sliding Window Code

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:
    $L \leftarrow ack$

Sender Ack Check

In correct operation, all values are in the range $L \ldots L + w$.

Currently, if the sender gets an ack outside this range, it copies it anyway. The sender is following, and the receiver is leading.

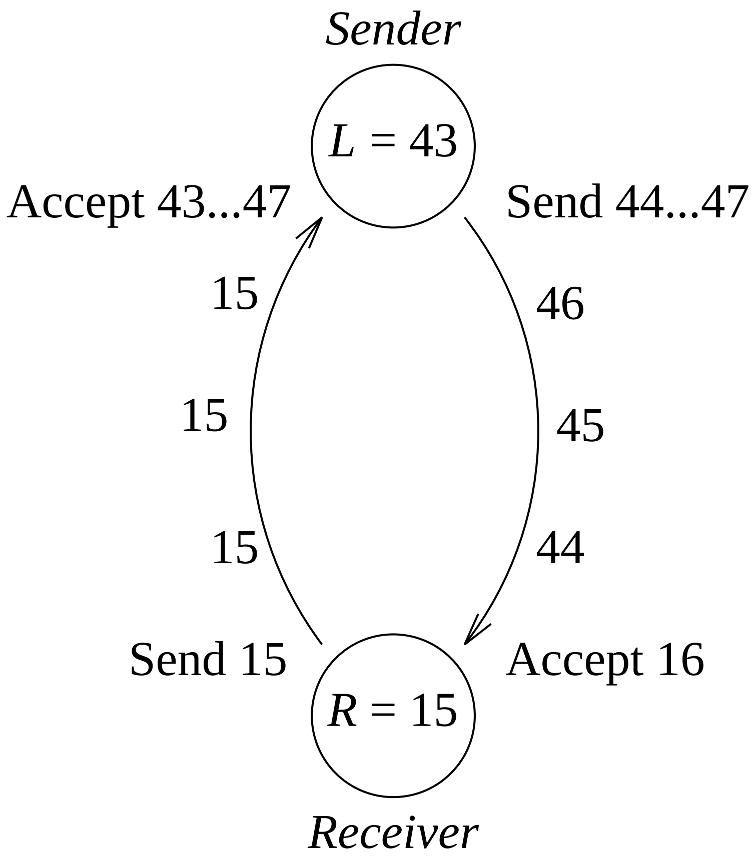
We want the sender to lead (so we can add multiple receivers later).

Sender Ack Check: Let the sender ignore "implausible" acks outside the range $L \ldots L + w$.

This is analogous to counter flushing, in which the leader checks incoming counters for equality to its own.

Receiver Window Adjustment

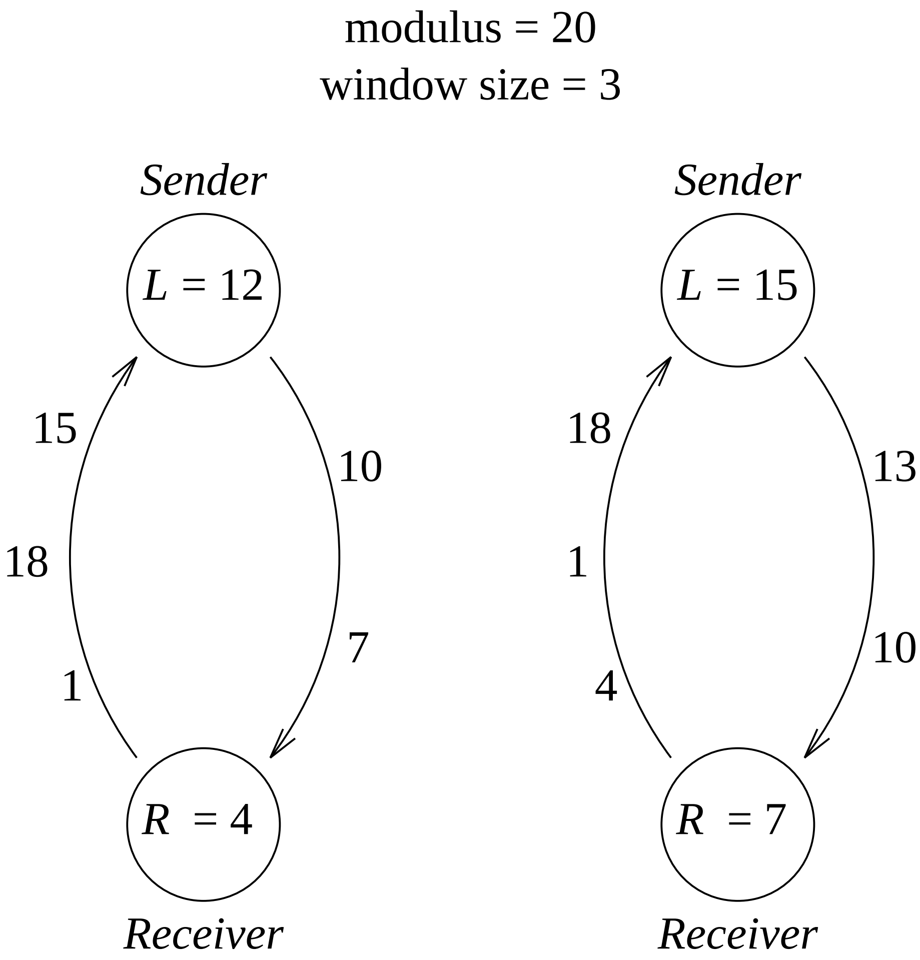
If we make the sender lead, we must make the receiver follow, lest we deadlock:



Receiver Window Adjustment: If the sender includes $L' = L$ in packets, the receiver should always find that R is in the range $L' \ldots L' + w$. If it isn't, the receiver should resync to the sender, setting R to the arriving sequence number.

Minimum Modulus

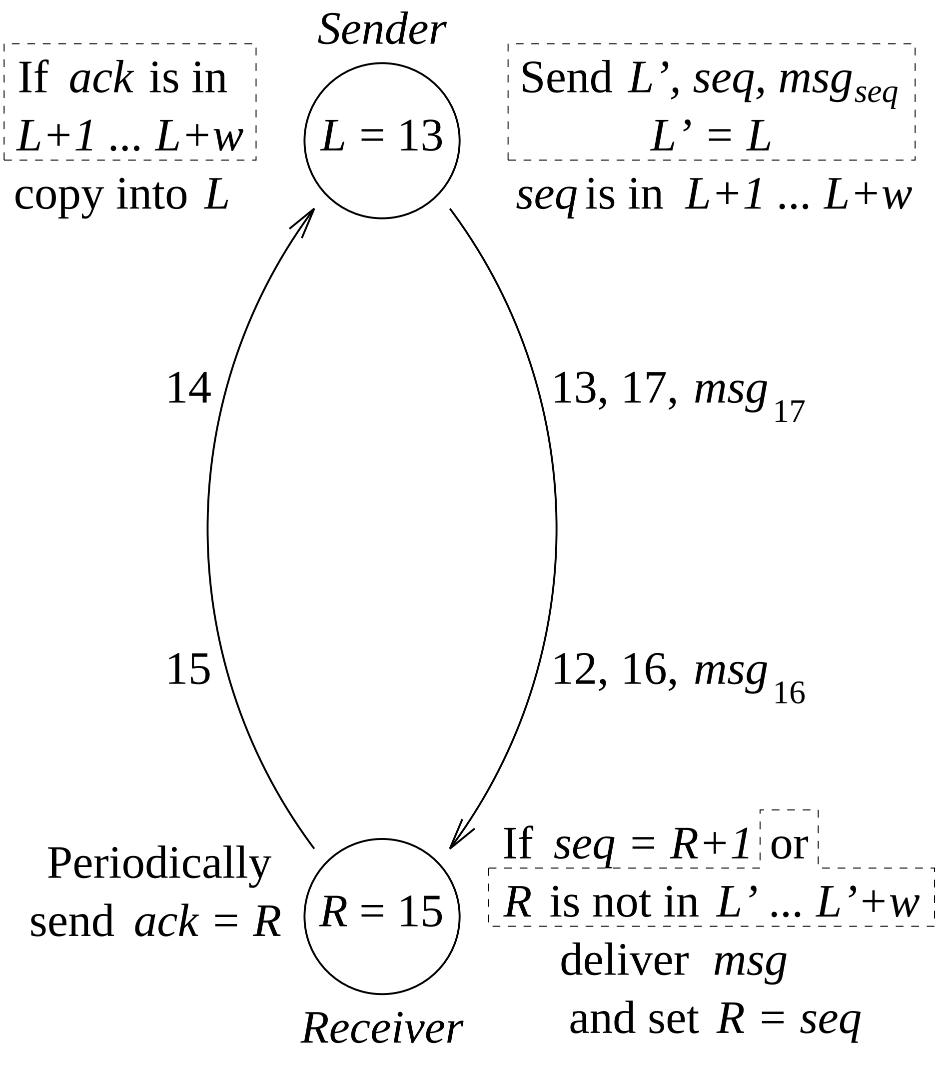
If the modulus is too small, the system can get stuck in a loop in which the receiver keeps adjusting, but never catches up to the sender.



Minimum Modulus: We require the modulus to be greater than the window size times Max, where Max is the maximum number of data packets, acks, and nodes that fit in the system.

Window Washing

The sender ack check, receiver window adjustment, and minimum modulus modifications to sliding window are together called *window washing*.



It can be shown that these modifications do not impair the correctness of the sliding window protocol.

Stabilizing Sliding Window Code

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:

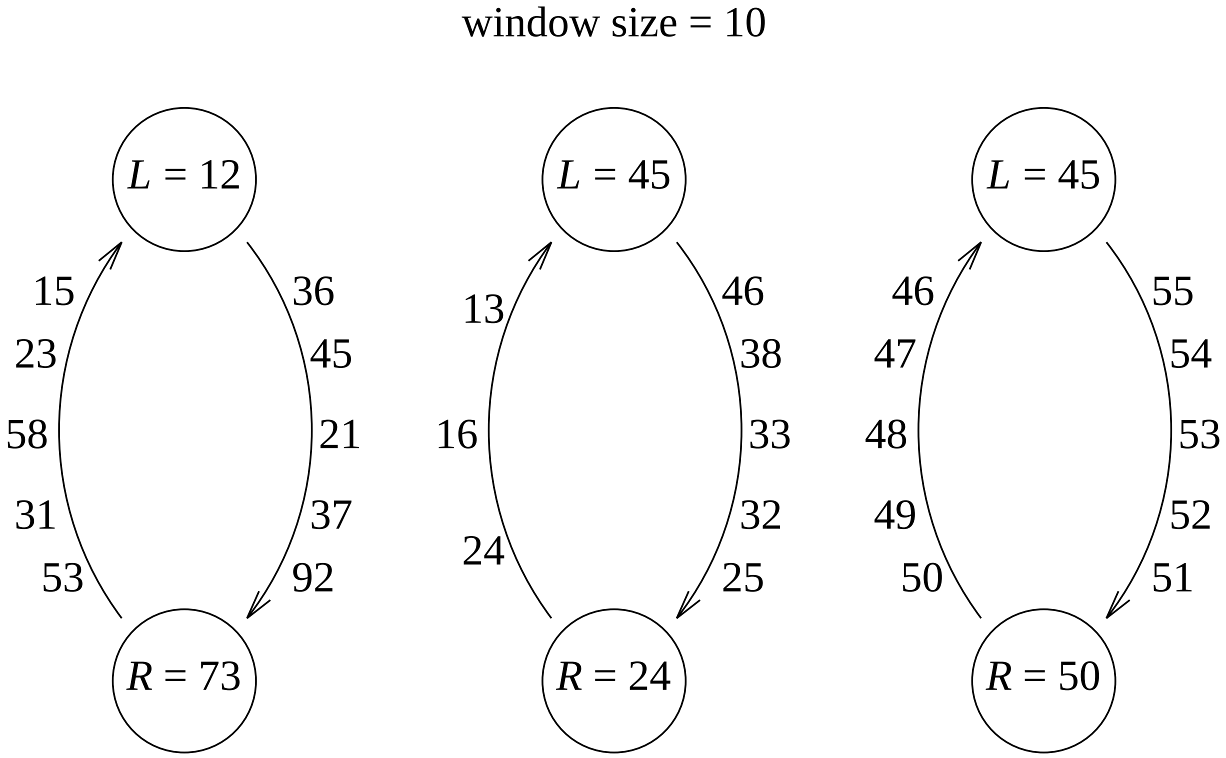
$\boxed{\textbf{if } ack \in [L+1, L+w] \textbf{ then}}$

$L \leftarrow ack$

$\boxed{\textbf{endif}}$

Stabilization of Sliding Window

Start from an arbitrary initial state, with $L = L_0$.



During the first round-trip delay, all initial garbage values are flushed, and L advances to L_f. Now all values in the system are in the range $L_0 \ldots L_f + w$. The sender ack check and minimum modulus insure that this range does not wrap all the way around the bounded integer space.

During the second round-trip delay, R catches up to L. The sender ack check insures that L remains stuck at L_f until it does, and the receiver window adjustment insures that R will indeed catch up.

Counter Flushing vs. 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

Adding More Receivers

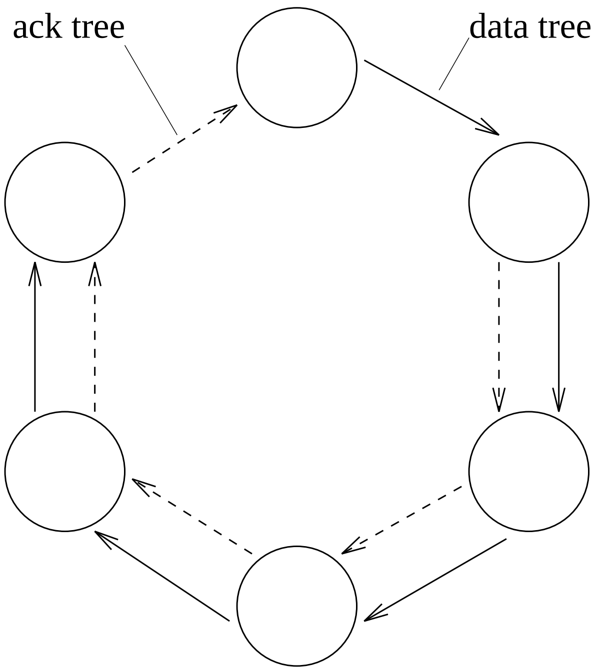
- One-to-many multicast is becoming common:
  - MBONE over IP
  - one-to-many ATM connections
  - Jon plans to add a reliable multicast extension to the Gigabit ATM switch.
- Pipelining is necessary for efficient use of bandwidth.
- Pipelining + reliability $\Rightarrow$ sliding window

One-to-Many Architecture

Two spanning trees, both rooted at the sender:

- *data tree* carries packets from sender to receivers
- *ack tree* carries acks from receivers to sender
- internal nodes act as relays

Topologies other than trees can be modeled—e.g. a ring:



The one-to-many protocol is several almost-independent 2-node sliding windows running between the sender and each receiver. Each 2-node protocol stabilizes within two round-trip delays, just as before.

One-to-Many Subtleties

Sender's Response to Acks:

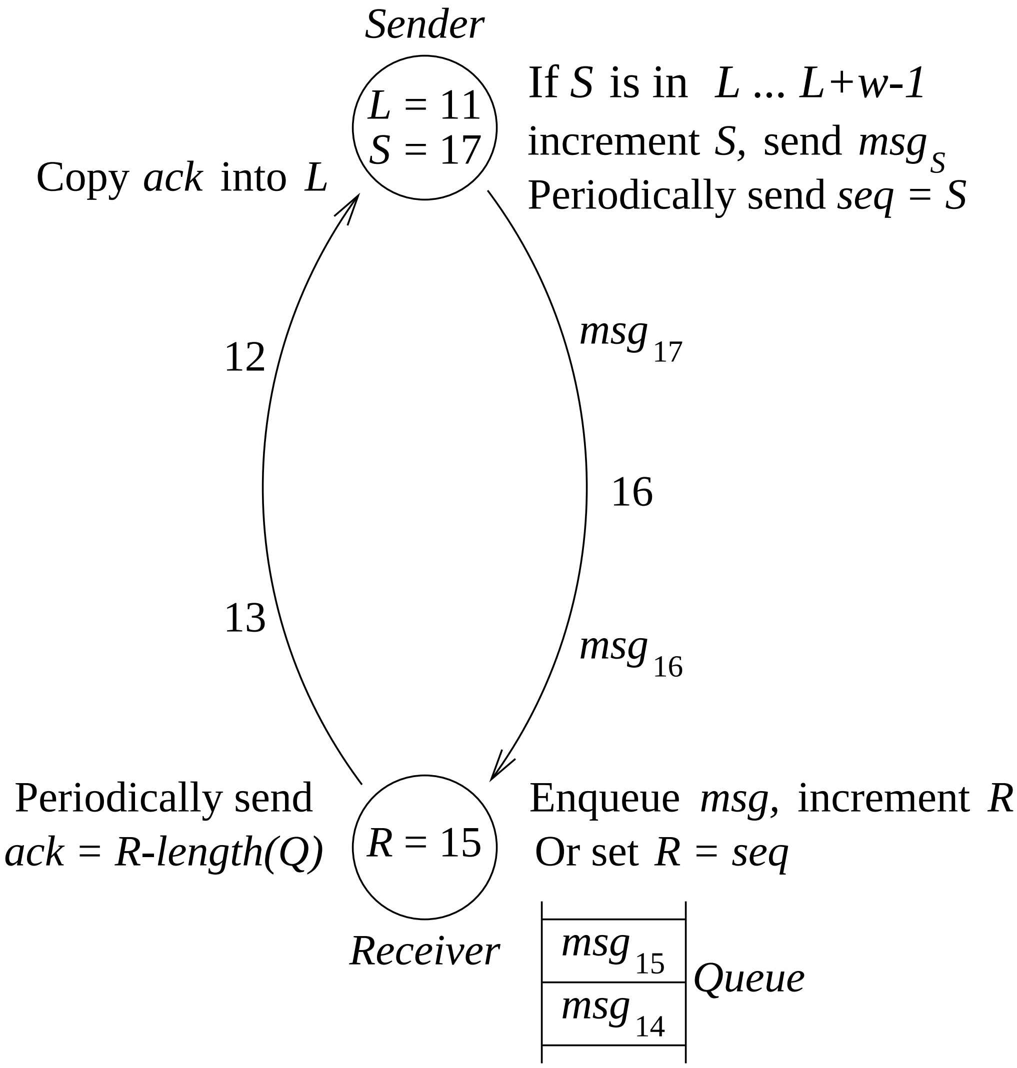
- Keep track of acks from each receiver separately.
- Advance L only if *all* the ack values are within $L+1 \ldots L+w$.
- Advance L to the minimum of the acks.

Optimization of Acks:

- A set of acks can be condensed into a fixed-size *digest* which records the range spanned by the set.
- Nodes can merge these digests on the way up the ack tree.

Flow Control

Minor modifications to the self-stabilizing sliding window yield self-stabilizing flow control, very similar to the Credit Update Protocol used in some ATM switches.



The sender ack check and minimum modulus are still present. The receiver window adjustment is too simplified to be visible (the receiver *always* adjusts).

Conclusions

- A new *theoretical* technique, window washing, which brings self-stabilization to pipelined broadcast and flow control.

- A new *practical* protocol, self-stabilizing one-to-many sliding window, which achieves reliable pipelined broadcast efficiently, and is highly fault-tolerant.

Jay Maisel, "Window Washer, Bogota, Columbia"