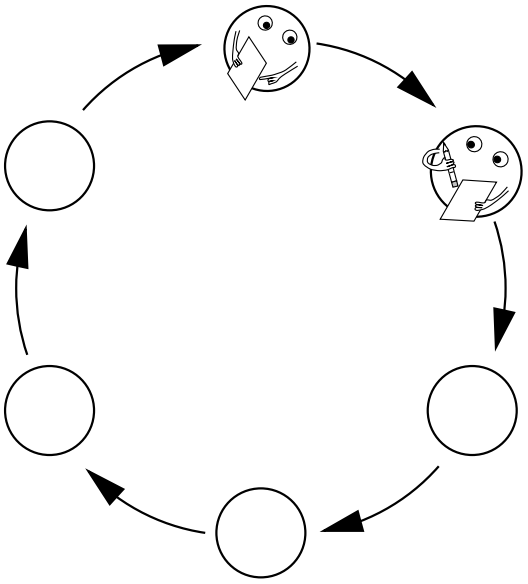
Self-Stabilization by Counter Flushing and Window Washing



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.

- **Counter Flushing for Real:** A modified FDDI MAC using counter flushing.

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

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

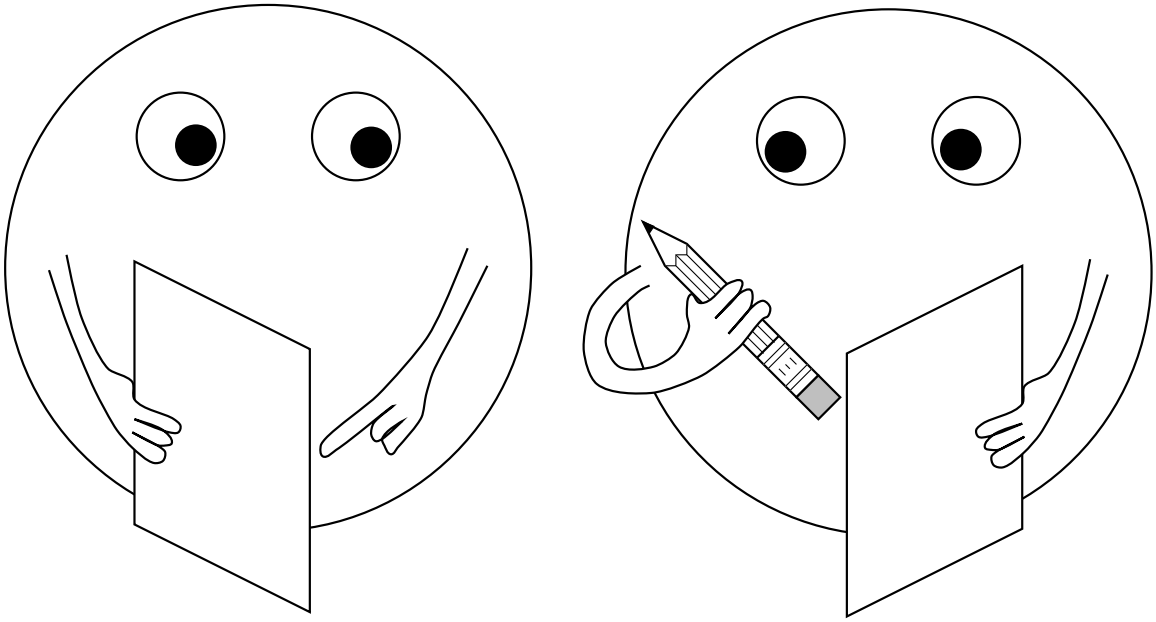
- **Flow Control** application of window washing.

Counter Flushing

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

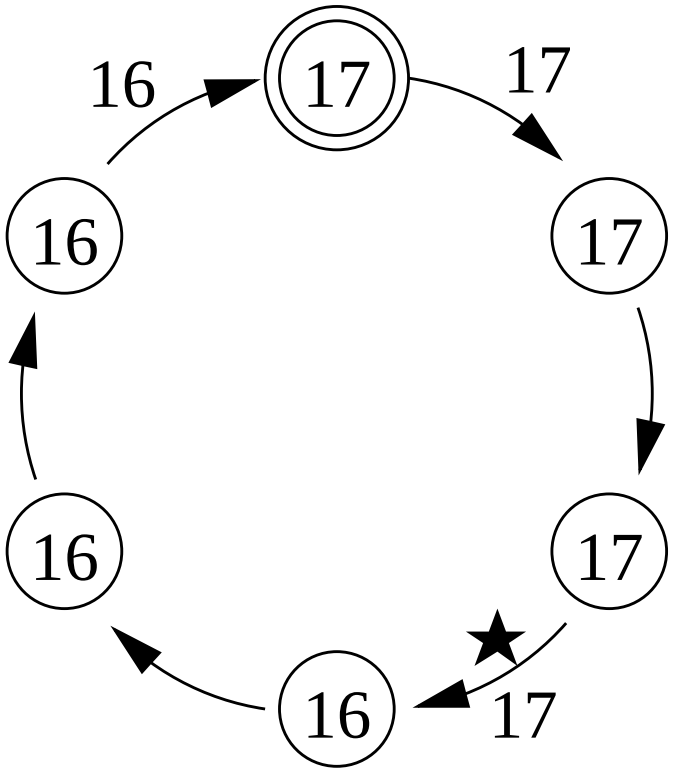
Nodes periodically inform neighbors of "where" in the execution they are.

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.

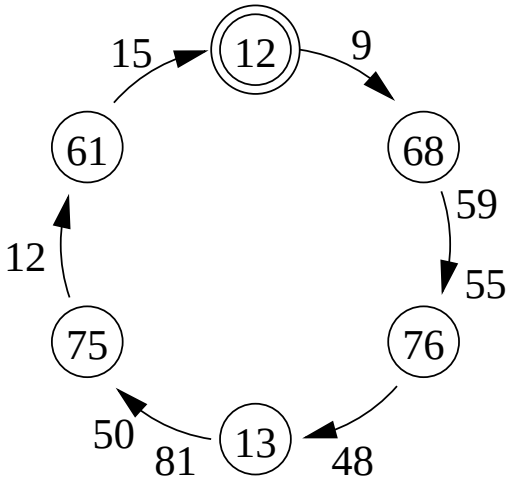
All nodes periodically send their counters.

Non-leaders always copy incoming counters.

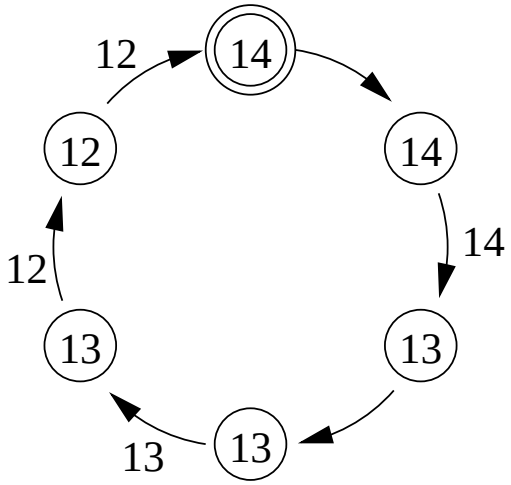
The leader increments when an incoming counter matches its own.

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

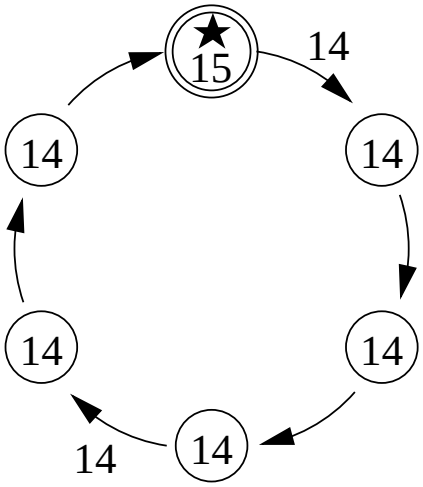
Stabilization of Token Ring



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 its own value has filled the ring (at most one more round-trip delay).

Counter Flushing for Real

Implementing counter flushing in a real-world token ring protocol (FDDI MAC) provides:

- Proof that the theoretical protocol can work in practice.
- Insight into the concerns of real token rings, not accounted for in theoretical counter flushing.
- Experience gauging the impact of counter flushing on performance and complexity.

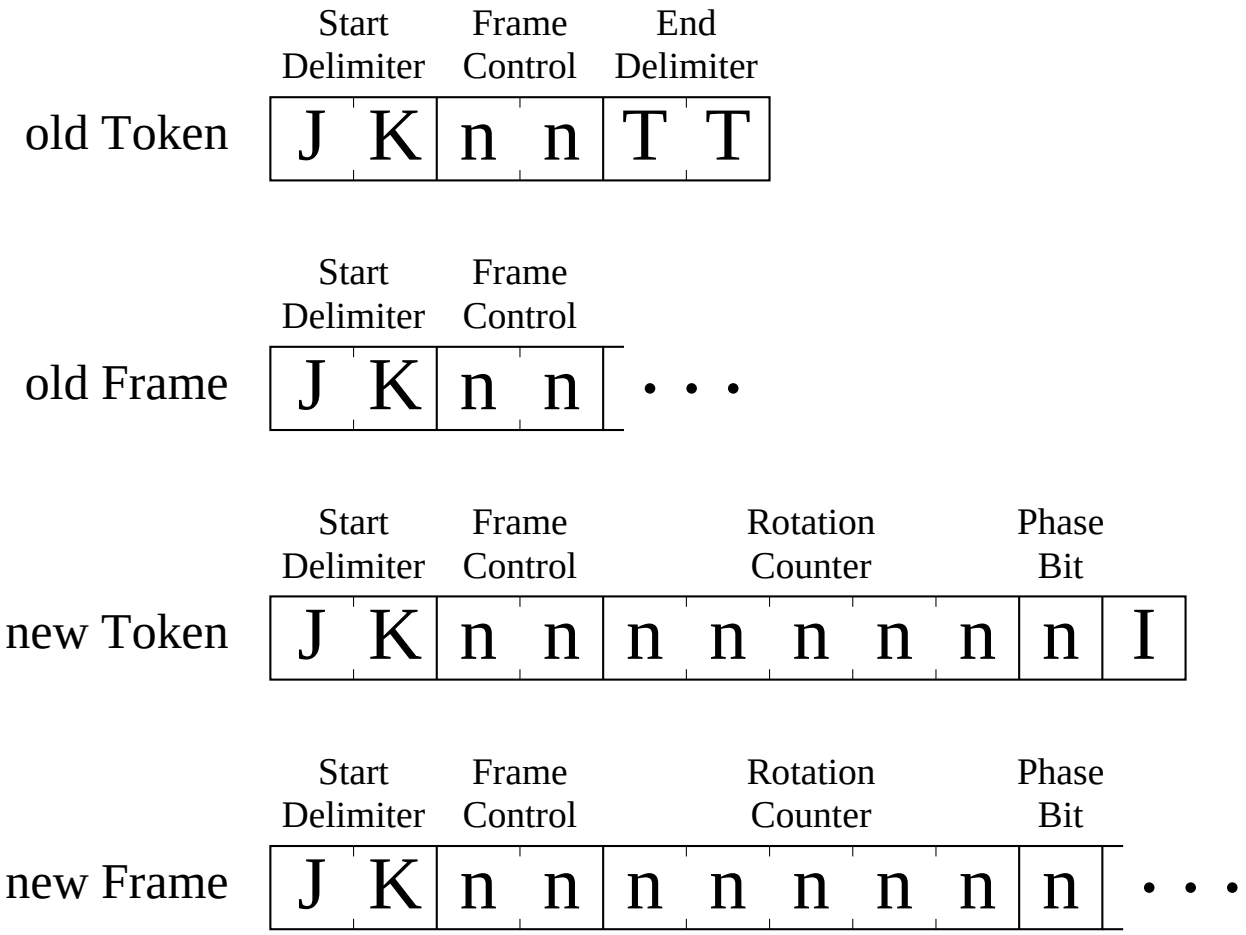
Concerns in Real Rings

- **Problem:** Periodic retransmission requires a buffer between input and output. Unacceptable latency.
- **Solution:** Only the Leader retransmits; other nodes let counter values pass through.

- **Problem:** To capture the token, nodes must modify the message as it passes through.
- **Problem:** Token rotation time can be *much* greater than ring latency. We want counter value to pass through even when token does not.
- **Solution:** Add a *phase bit* which can be cleared to prevent a message from conveying the token.

- **Problem:** How to choose the Leader?
- **Solution:** *(not implemented)* Self-stabilizing leader-election algorithms exist. For now, we use the Claim Token process.

Message Formats



Tokens look like, and are treated like, Frame fragments.

Token or Frame with set Phase Bit *may* convey the token.

Message Creation

Each node contains an internal rotation counter and phase bit.

To relinquish the token quickly:
Copy internal rotation counter and phase bit into a new Token.

To send data while holding the token:
Copy internal rotation counter into a new Frame, but leave the Phase Bit clear.

Non-Leader Behavior

Always copy the incoming Rotation Counter and Phase Bit into the internal register (if error-free).

If register changes and phase bit is now set, optionally capture token by clearing the Phase Bit in the outgoing message.

When there is nothing to send or repeat, emit Idle Symbols (I).

Leader Behavior

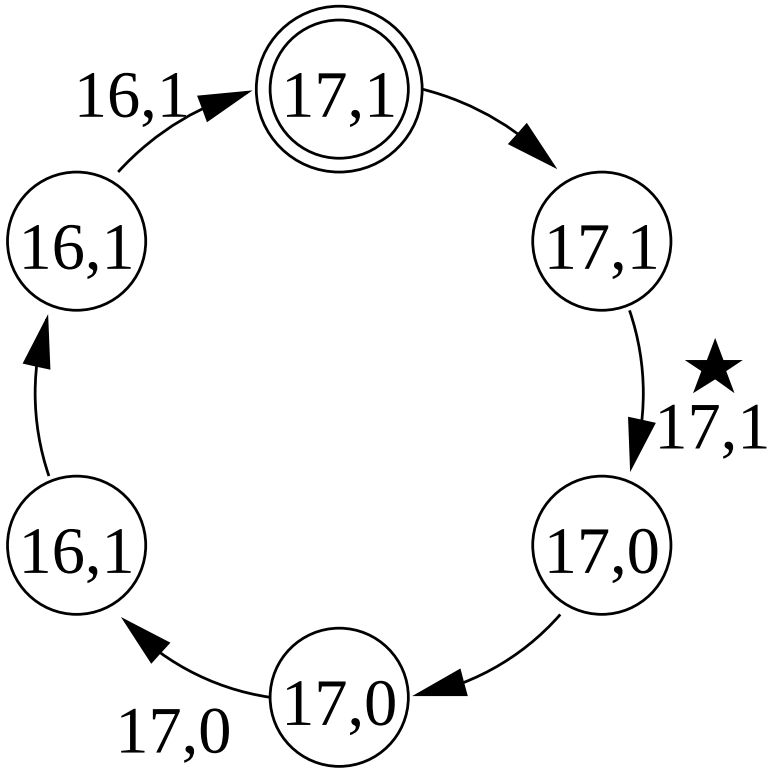
Always copy the internal rotation counter and phase bit into the outgoing message.

If that caused no change, have token, increment internal rotation counter.

Leader's internal phase bit is *always* 1.

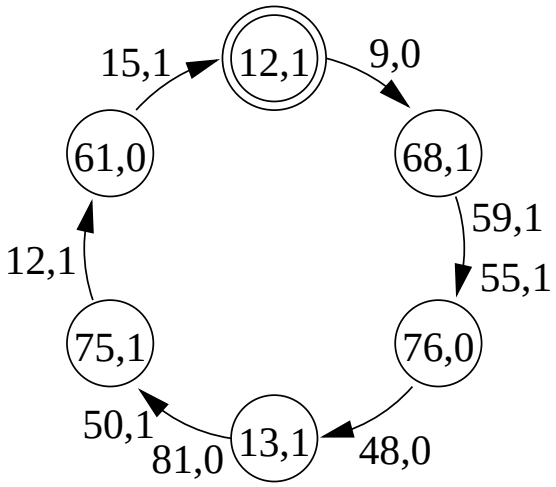
When there is nothing else to send or repeat, send Tokens. (Requires buffer big enough to hold one Token.)

Ring Operation

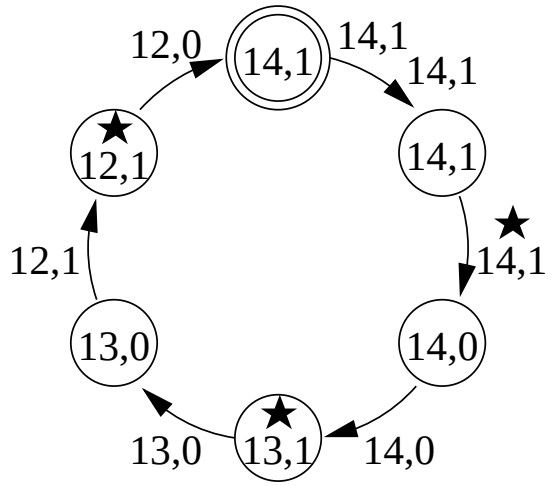


Three bands: New counter value propagates faster than the token.

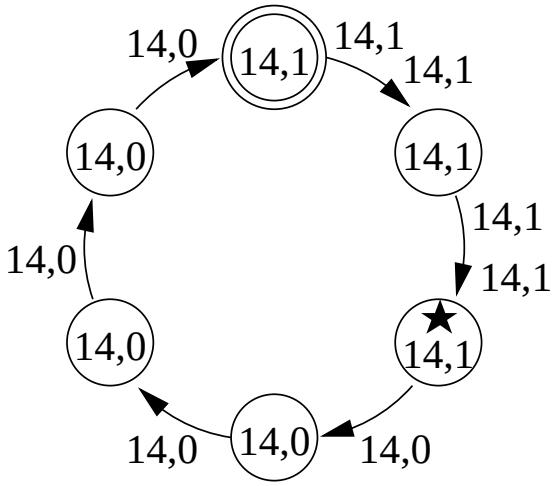
Ring Stabilization



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



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



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

Costs of FDDI Modification

- **Longer Token** (by 5 symbols) costs 200 ns per capture. For typical TTRT of 10 ms and maximal ring of 1000 stations, bandwidth reduced up to 2%, proportionately less for smaller rings.

- **Longer Frames** (by 6 symbols) costs 240 ns per Frame. For payload lengths of 512 bytes, bandwidth is reduced 0.6%.

- **Complexity** has increased somewhat, but not much compared to the parts of the FDDI MAC left untouched.

Benefits of FDDI Modification

- **Faster recovery** from lost tokens.

  old: TVX timeout plus ring latency
  (2.5 ms + up to 1.6 ms)

  new: Frame fragment time
  (0.36 ms max, 0.02 ms typical)

- **Robustness** from state corruption. E.g. old protocol might never recover from multiple tokens.

- **Simplicity:** TVX can optionally be eliminated.

FDDI Simulation

Simulator performs virtually all actions of the new FDDI MAC protocol, down to the symbol level.

Implementation led to a few tweaks of the spec, e.g. the need for an Idle Symbol (I) at the end of a Token.

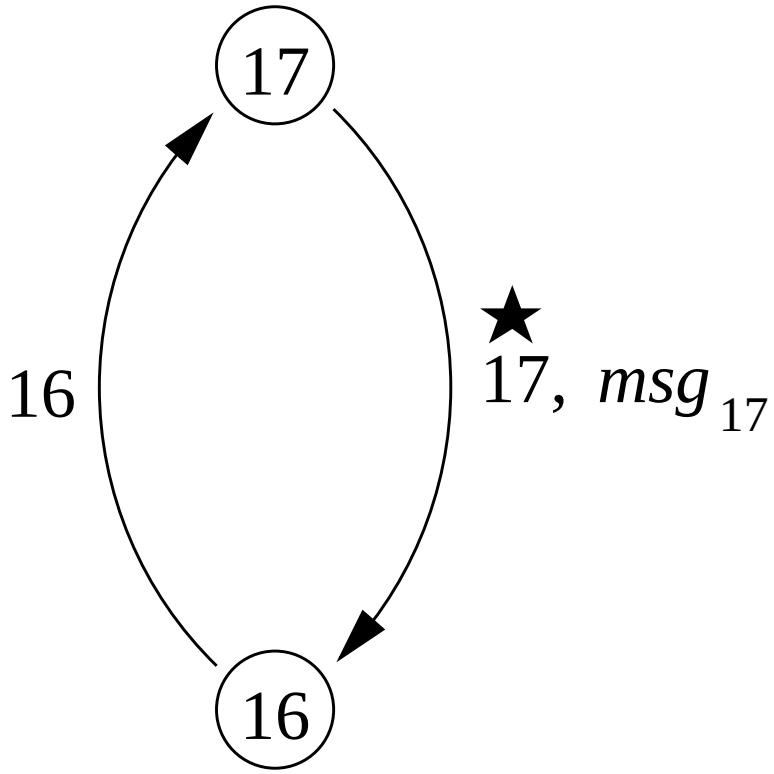
Results as expected:

- Recovery from a lost token is very fast.
  (1.0–2.2 μs in tests,
  360 μs for maximal 4500-byte Frames)

- Recovery from an arbitrary state typically takes just over one ring latency.
  (52 μs for a 10 km fiber ring)

From Token Ring to Stop & Wait

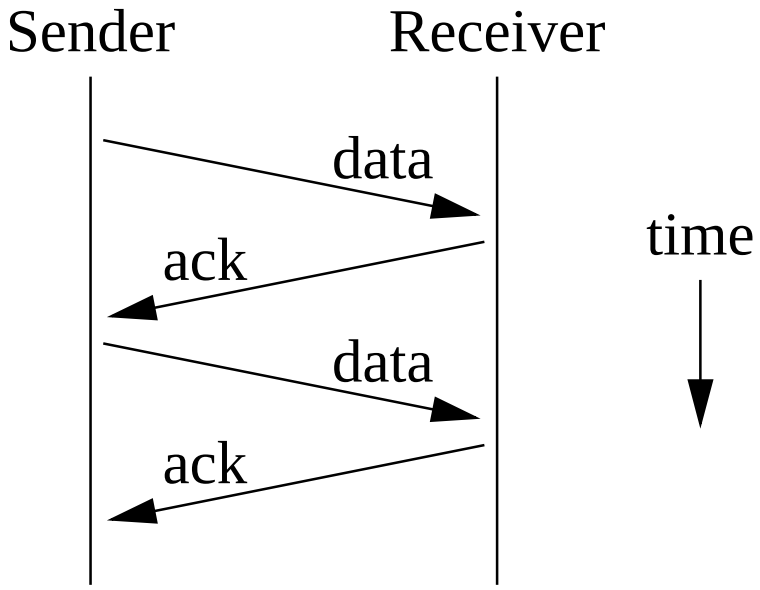
two-node token ring + piggybacked data = stop & wait



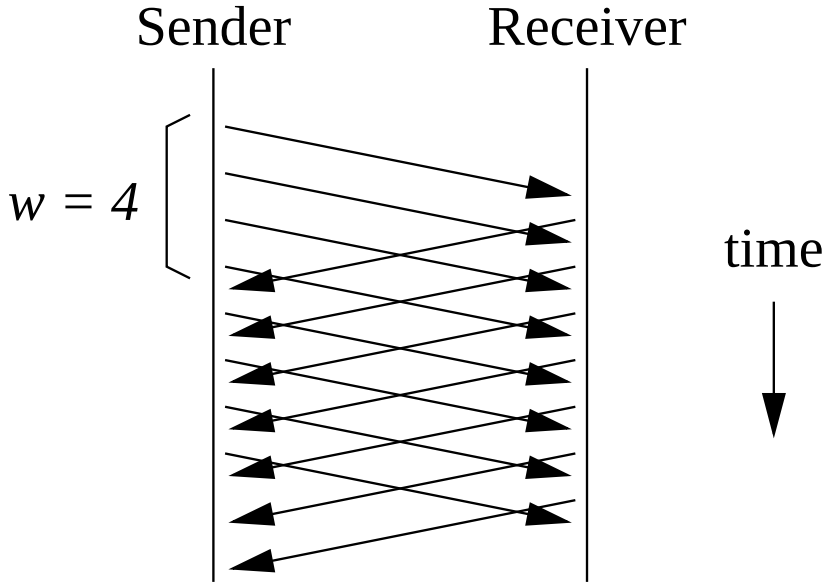
The stop & wait protocol can be extended to the *sliding window* protocol to allow pipelining.

Stop & Wait vs. Sliding Window

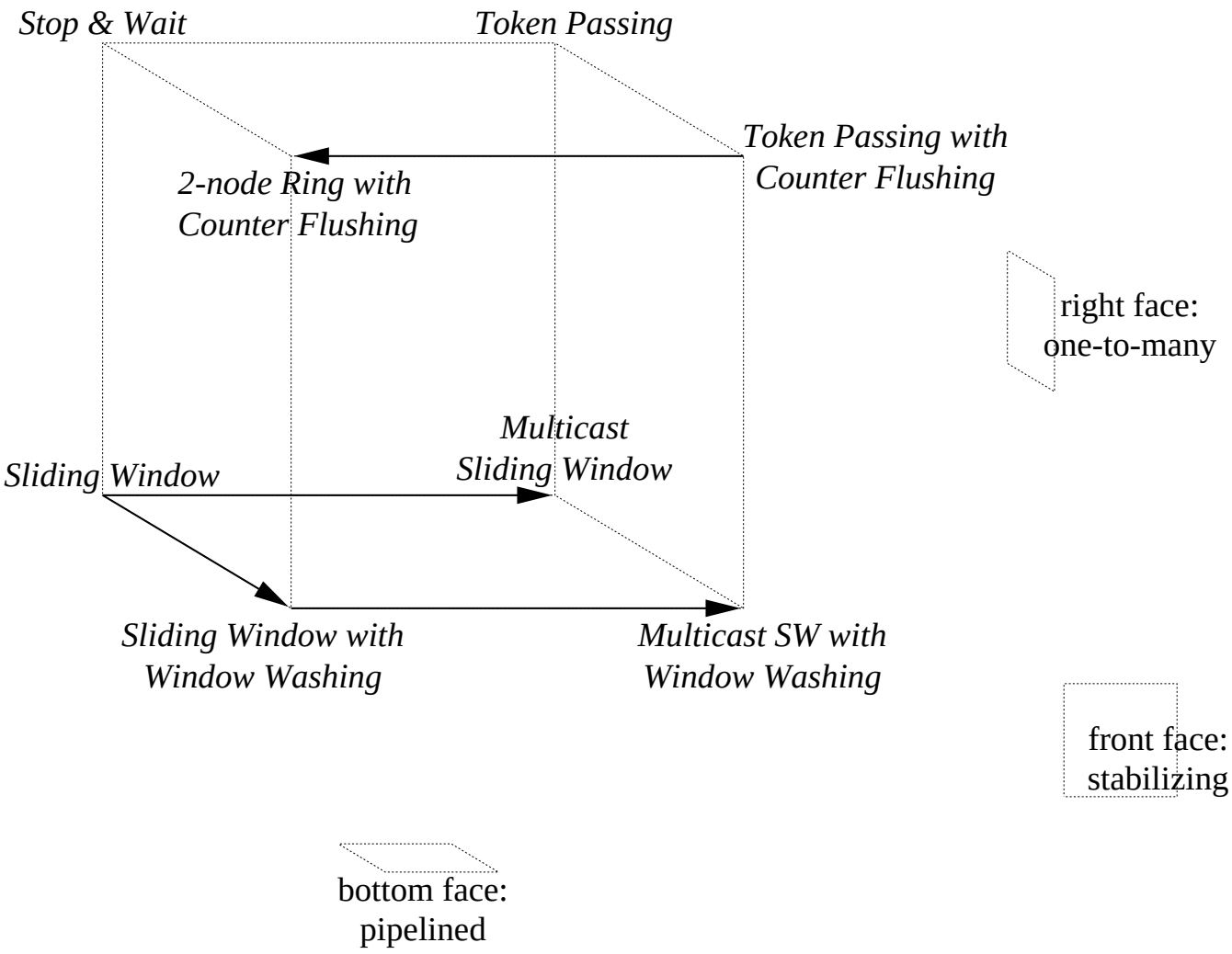
With stop & wait, each message requires one round-trip delay.



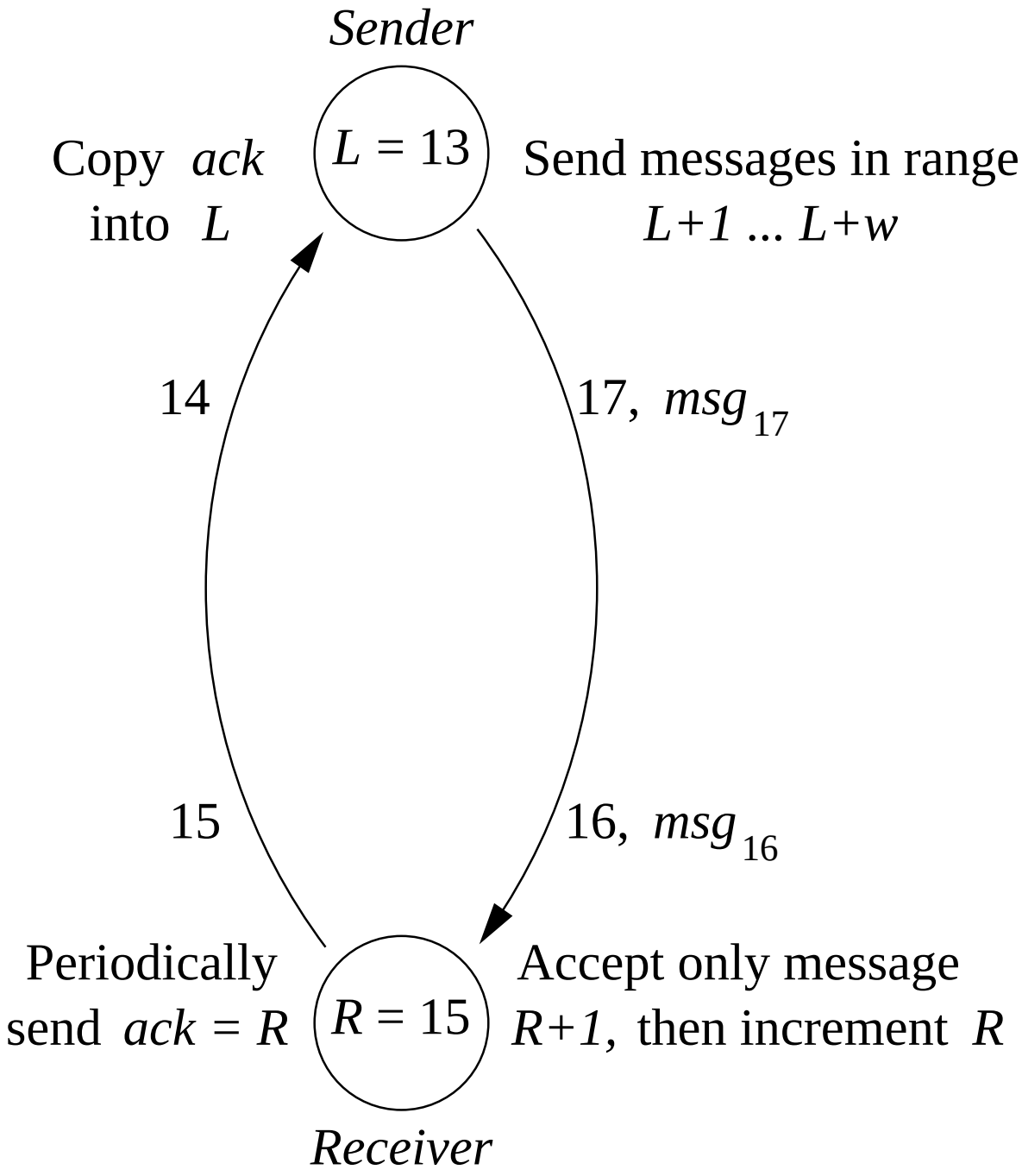
With sliding window, each round-trip delay is shared by w messages.



Protocol Taxonomy



Sliding Window Protocol



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.

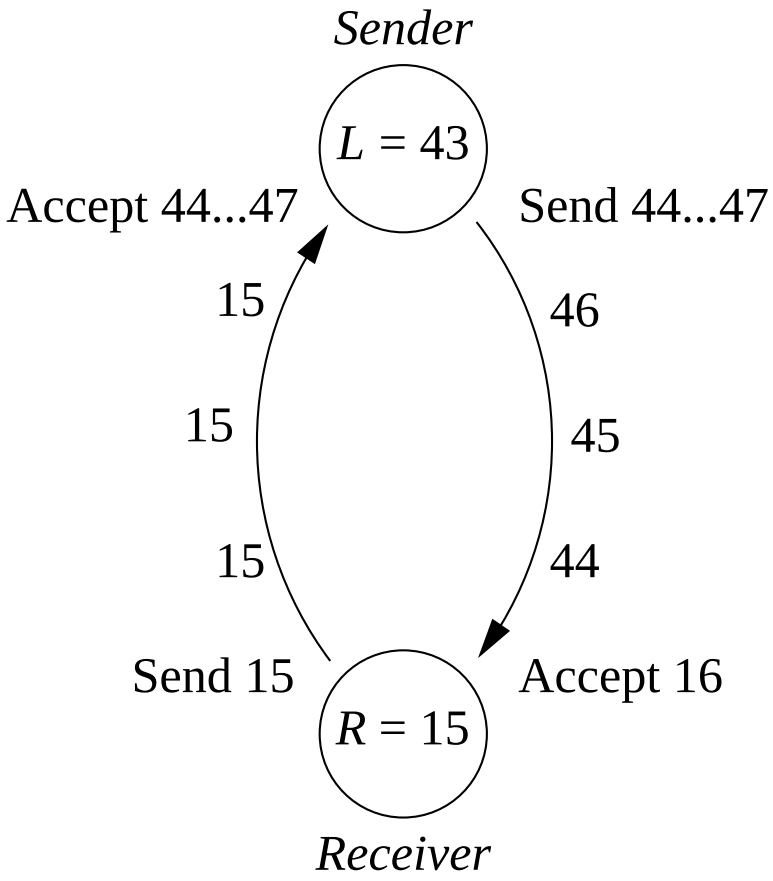
We want the sender to lead (so we can add multiple receivers later).

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:

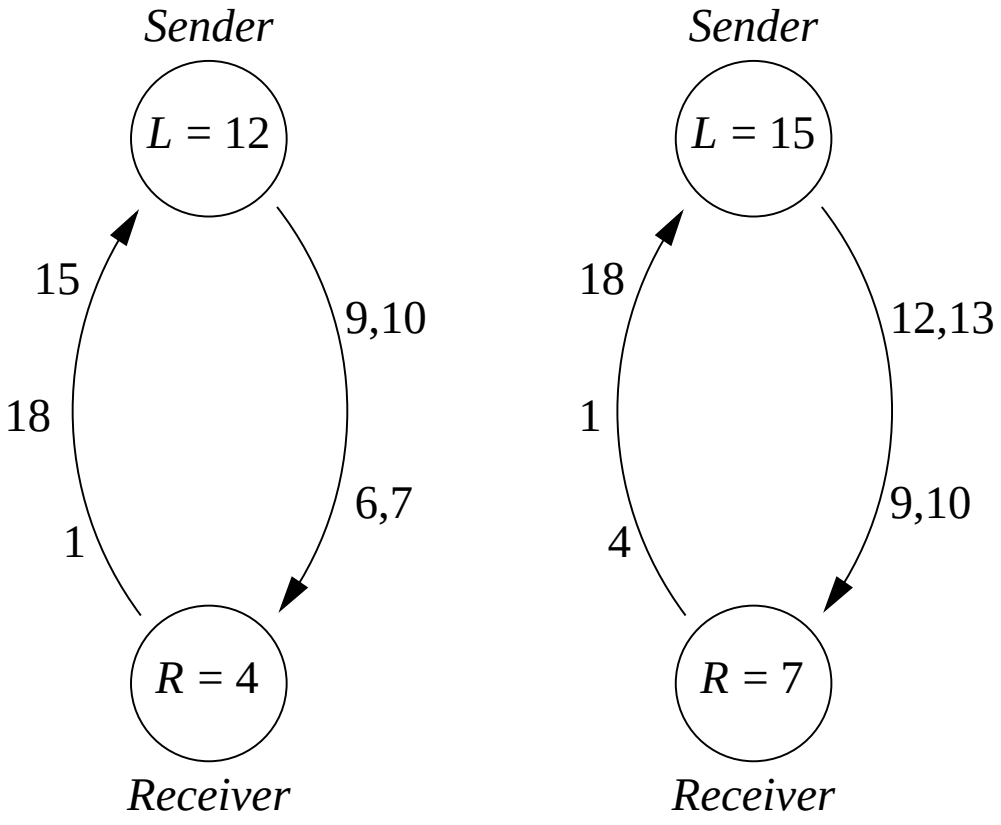


Receiver Window Adjustment: If the sender includes $L' = L$ in packets, the receiver should always find that R is in the range $L' \ldots L' + w$. If it isn't, the receiver should resync to the sender, setting R to the arriving sequence number.

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.

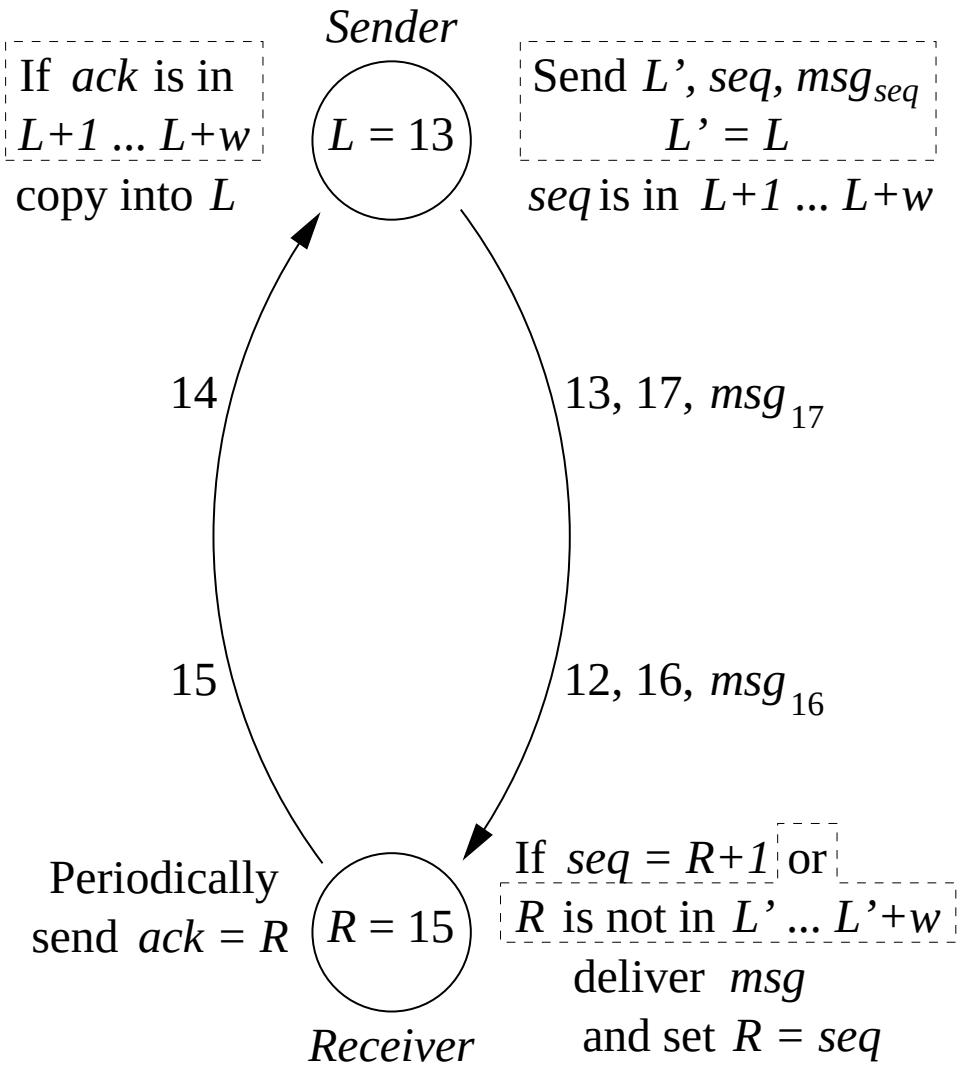
modulus = 20
window size = 3



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

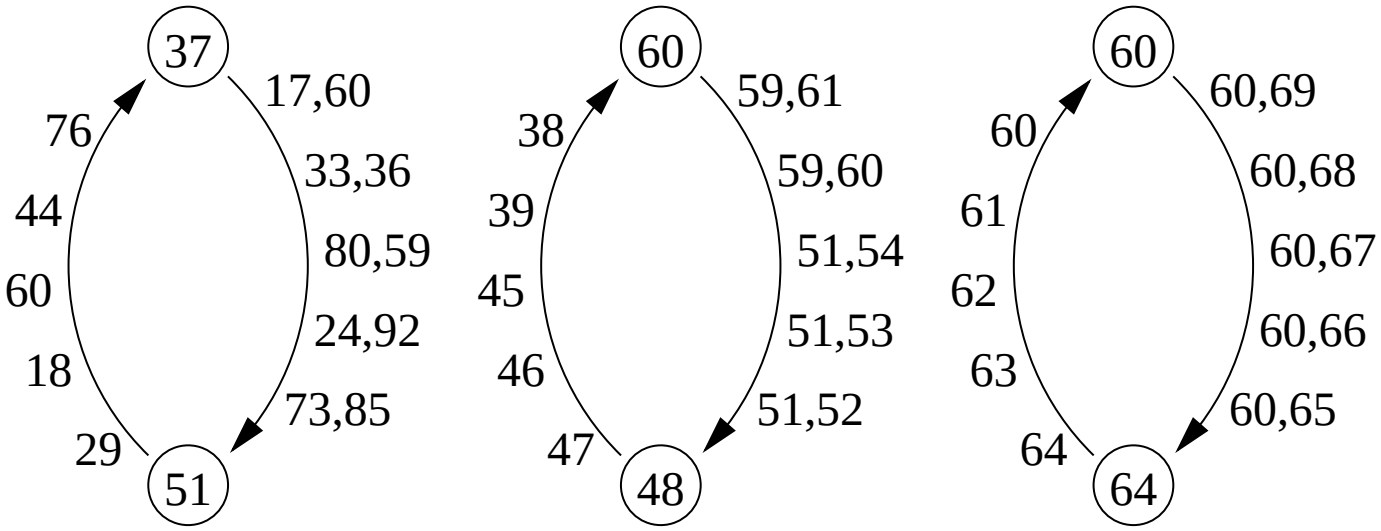
The sender ack check, receiver window adjustment, and minimum modulus modifications to sliding window are together called *window washing*.



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

Stabilization of Sliding Window

Start from an arbitrary initial state, with $L = L_0$ and window size = 10.



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

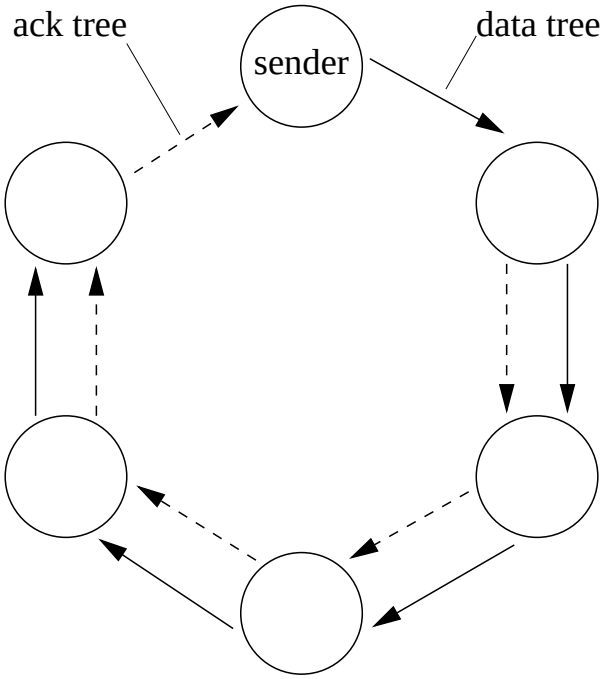
- One-to-many multicast is becoming common:
  - MBONE over IP
  - one-to-many ATM connections
  - Jon plans to add a reliable multicast extension to the Gigabit ATM switch.
- Pipelining is necessary for efficient use of bandwidth.
- Pipelining + reliability $\Rightarrow$ sliding window

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:

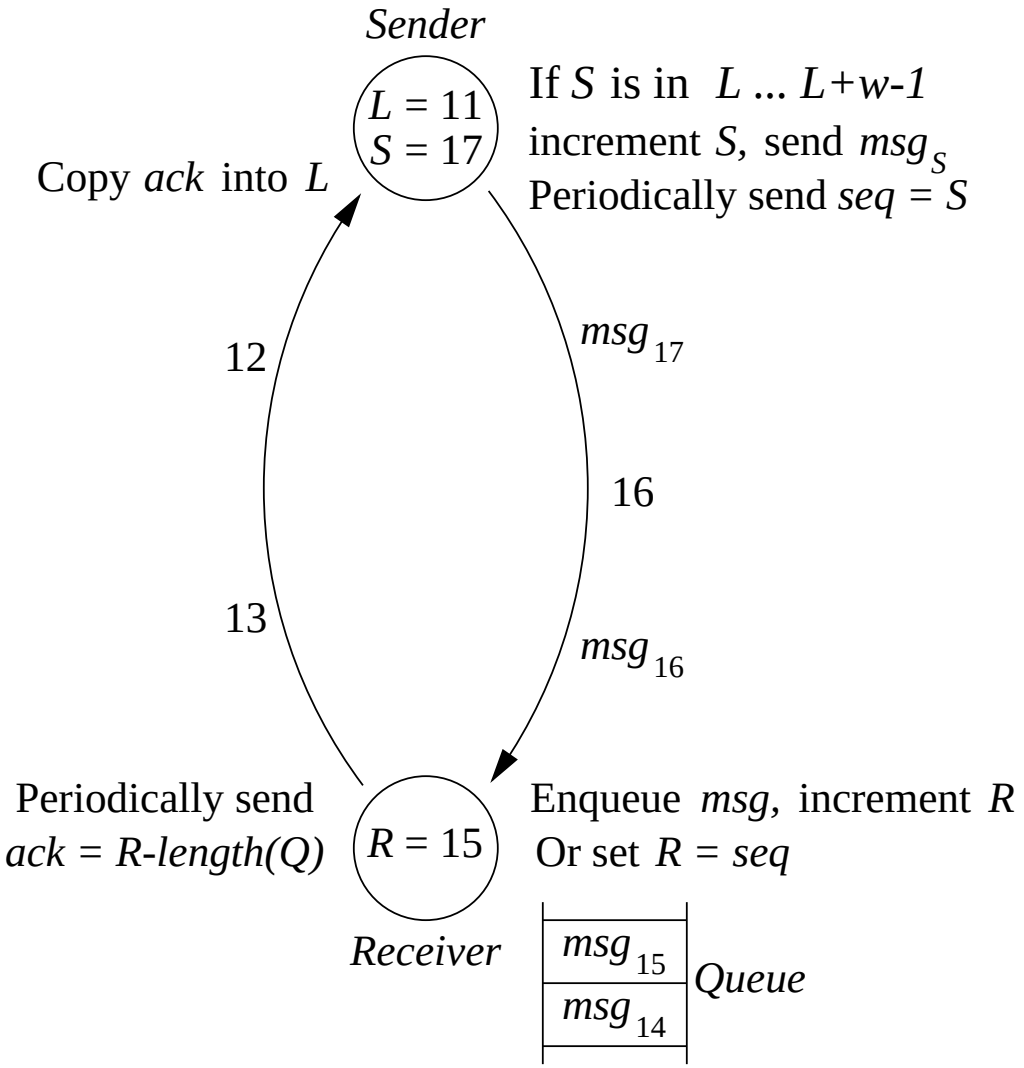
- Keep track of acks from each receiver separately.
- Advance L only if *all* the ack values are within $L+1 \ldots L+w$.
- Advance L to the minimum of the acks.

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

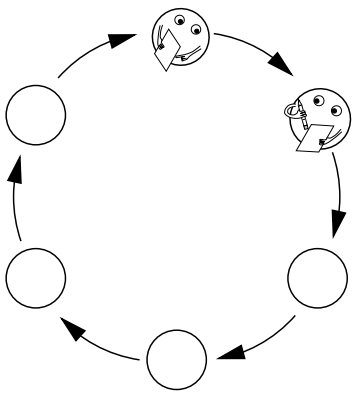
Minor modifications to the self-stabilizing sliding window yield self-stabilizing flow control, very similar to the Credit Update Protocol used in some ATM switches.



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

- Experience applying counter flushing to a real-world token passing protocol.

- 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.