Self-Stabilizing Multiple-Sender / Single-Receiver Protocol

Karlo Berket, Ruppert Koch

Department of Electrical and Computer Engineering, University of California, Santa Barbara, CA 93106, karlo@alpha.ece.ucsb.edu, ruppert@alpha.ece.ucsb.edu

Abstract. We present a new self-stabilizing protocol for many-to-one multicasting of messages. It is based on the window washing protocol of Costello and Varghese which uses positive acknowledgments for received messages. The assumed model uses a single queue at the receiver's side taking all the messages sent by N senders. The protocol provides flow control independently for every sender by dividing the queue into N logical queues. To assure good performance for bursty traffic, the share of the queue a sender is holding is adapted dynamically to the amount of traffic emitted by the sender. A proof of correctness and an upper bound for stabilization are given.

1 Introduction

In a time of increasing software complexity, protocols must be designed in a robust manner. They should be able to cope with unforeseen erroneous situations. Since Dijkstra [3, 6] introduced self-stabilization, much work has been done on designing protocols that use this property to guarantee return to a correct state in case of transient errors [1, 5].

For low-level communication protocols, self-stabilization turns out to be particularly useful. Unreliable communication causes errors that must be handled by the protocol. Gouda and Multari [4] and Spinelli [7] developed self-stabilizing two-node sliding-window protocols. In [2] Costello and Varghese introduced a self-stabilizing sliding-window protocol called window washing. Window washing deals with one-to-one and one-to-many communication.

Sliding-window protocols [8] for one-to-one links are straightforward. The main problem — protecting the receiver's queue from overflowing with messages — can be solved by controlling the number of messages the sender is allowed to transmit. An implementation of many-to-one communication must address the problems that arise when several sources are feeding the receiver queue. Splitting up the receiver buffer into N buffers (where N equals the number of senders) cannot usually be done since buffer management is a part of the network interface itself.

Another approach is to divide the buffer logically into N parts by assigning the window size of the senders so that the sum of all of the window sizes is less than or equal to the size of the receiver buffer. Using fixed window sizes is easy to implement but turns out to be wasteful if the traffic is bursty. A way to achieve better performance is to assign flexible window sizes. Senders emitting more messages than others can use a wider window. The sum of all of the window sizes must remain constant.

The protocol proposed in this paper achieves this by permanently checking the amount of data each sender is sending. The assigned window sizes are piggybacked in the acknowledgments. It can be seen that, by applying strict assignment rules and periodically resending some of the data, the protocol becomes self-stabilizing. Loss of messages, loss of acknowledgments, arbitrarily assigned window sizes, and inconsistent views of the system can be overcome without any additional control within a well-defined period of time. This makes our protocol both flexible and robust.

The paper is organized as follows. Section 2 briefly describes window washing and explains the mechanisms by which self-stabilization is achieved. Section 3 introduces the window size adjustment protocol for a one-to-one protocol. Section 4 expands the protocol for many-to-one communication. A worst-case analysis and a proof of correctness are given. Section 5 provides simulation results. Section 6 states our conclusions.

2 Window Washing

The window washing protocol proposed by Costello and Varghese [2] is a sliding-window protocol that can be used to impose flow control in one-to-one and one-to-many setups. Here we consider the one-to-one setup as it is given in Figure 1.

Every message is tagged with a sequence number seq. Each message is retransmitted periodically until the sender receives an acknowledgment for it. If the sender window size is w, the sequence number of messages sent is in the range $[L+1, L+w]$. L denotes the lower window edge and is attached to every message.

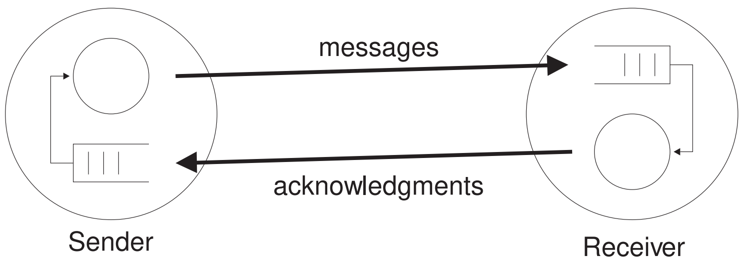

Fig. 1. One-to-one setup. The sender sends messages to the receiver, and the receiver sends acknowledgments back. The window washing protocol provides flow control and reliable message delivery. The channels are FIFO channels.

If the receiver receives a message with the sequence number seq, it checks if seq is equal to $R+1$ where R is the sequence number of the last message it has received. If this is the case, it increments R. If this is not the case, it discards the message. However, if seq is not in the range of $[R+1, R+w]$, the receiver accepts the message and copies the value seq from the message header into R.

After a valid message is received, R is sent as an acknowledgment. It acknowledges the message with sequence number R and all previous messages. Like all unacknowledged messages, the last ack is retransmitted periodically to protect the system from data loss. The sender accepts an ack if R is in the range $[L+1, L+w]$ and sets its lower window edge L equal to R. Figure 2 shows the protocol.

SendData($L, seq, message$) /* Sender emits message */
    Precondition: $seq \in [L+1, L+w]$

ReceiveData($L, seq, message$) /* Receiver absorbs message */
    if ($R \notin [L, L+w]$ or $seq = R+1$) then
        $R = seq$
        deliver message
    endif

SendAck(ack) /* Receiver emits ack */
    Precondition: $ack = R$

ReceiveAck(ack) /* Sender absorbs ack */
    if ($ack \in [L+1, L+w]$) then
        $L = ack$
    endif

Fig. 2. Window washing protocol. Variables used are window size w, lower window edge L, sequence number seq of current message, number R of last valid message received by receiver, and acknowledgment number ack.

Costello and Varghese show that the protocol is self-stabilizing if $seq \in [0, M]$ with M not smaller than $w \cdot c_{max}$, where c_{max} is the maximum number of messages that can be in the system at any point in time.

3 Window Size Adjustment

Whereas the window washing protocol works fine for one-to-one and one-to-many communication, many-to-one communication is a different kettle of fish. In this form of communication, many senders are writing to the same receiver queue. The queue is divided into N logical sub-queues by assigning window sizes that can be adjusted to the amount of traffic a sender is creating. FIFO channels are assumed.

We introduce our protocol in two steps. First, we describe a one-to-one protocol based on window washing that includes the feature of window size adjustment. In a second step, the new one-to-one protocol is expanded to a many-to-one protocol. It is proven that both protocols are self-stabilizing and a worst case analysis are given.

3.1 Description of the Protocol

The protocol consists of two parts: window washing is responsible for flow-control and reliable transfer of messages and acknowledgments, and the window size assignment adjusts the window size used by the sender.

On the sender's side only minor changes need to be made. In addition to the lower window edge L and a sequence number seq, a message is tagged with the sender's current window size w_s. The acknowledgment now contains the sequence number and the new window size. Every time an acknowledgment arrives — valid or not — the sender overwrites its old value of the window size with the new one. Apart from these modifications all senders run the window washing protocol.

Attaching the new window sizes to the acknowledgments has the advantage that no additional messages are needed. This reduces the traffic and, much more importantly, allows distribution of the new window sizes without worrying about loss of these messages. However, there is a price to pay. The assignment is less flexible: The sender is notified of a change in its window size only if it receives an acknowledgment.

The window size can be increased in arbitrary steps without informing the corresponding sender. This is not true for decreasing the window size. We can reduce the window size of the sender only if the sender receives an acknowledgment. The reduction must not be larger than the number of messages that are acknowledged with the ack. Imagine that the receiver attaches a new window size to an ack that is equal to the old size diminished by more than one. At the time the sender receives the ack, it has already sent all messages it was allowed to send according to its old window size. Buffer overflow can occur.

Decrementing the window sizes in steps of one protects the receiver buffer from overflow even if acknowledgments get delayed or lost. Since the sender cannot move its lower window edge L, no more than the allowed number of messages can be generated.

Pseudo-code for the protocol is given in Figures 3.

3.2 Proof of Correctness

The main idea of the proof of correctness is to split up the protocol into two independent parts: the window washing protocol and the window size assign-

```
SendData( L, w_s, seq, message )              /* Sender emits message */
        Preconditions: seq ∈ [L + 1, L + w]

ReceiveData( L, w_s, seq, message )           /* Receiver absorbs message */
        if (R ∉ [L, L + w_s] or seq = R + 1) then
            R = seq
            deliver message
            AdjustWindowSize()
        endif

SendAck( w, ack )                             /* Receiver emits ack to j */
        Preconditions: ack = R
        CheckWindowSize()

ReceiveAck( w, ack )                          /* Sender absorbs ack */
        w_s = w
        if (ack ∈ [L + 1, L + w]) then
            L = ack
        endif

CheckWindowSize()                             /* check for correct window size */
        if (w ∉ [w_min, w_max]) then
            w = w_min
        endif

AdjustWindowSize( newsize)                    /* adjusts window size */
        if (w_min ≤ newsize ≤ w_max) then     /* check if new size is in
                                                 valid range */
            if (w − 1 ≤ newsize) then         /* check if window size is not
                                                 decreased too much */
                w = newsize
            endif
        endif
```

Fig. 3. Send, receive, check, and adjust routines for the many-to-one protocol: Send and receive are similar to the window washing protocol .

ment protocol. It can be shown that both parts fulfill the requirements of self-stabilization. Then the two parts can be merged and it can be proven that the protocol works correctly. Figure 4 illustrate the separation into two independent parts.

Although the window-assignment protocol logically rests on top of the window washing protocol, the order is reversed in this proof. It is easy to see that there is no round trip flow of information in the window-assignment part. The

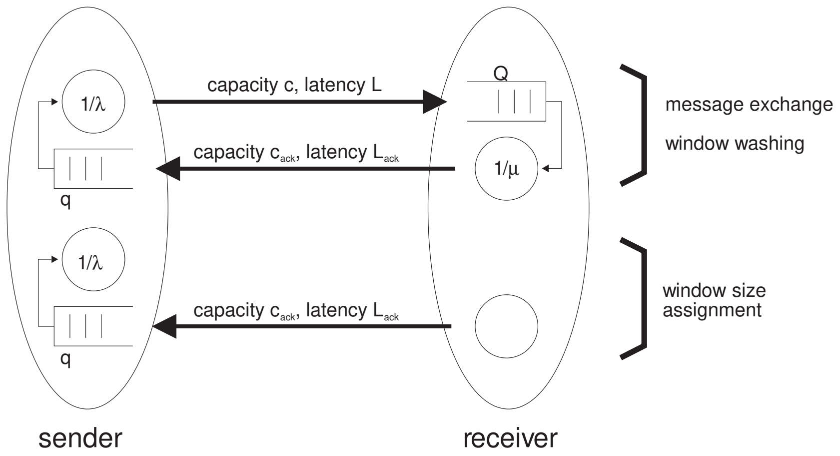

Fig. 4. Logical structure of the protocol. The upper sender-receiver pair represents the window washing protocol, the lower pair the window size assignment protocol. Q denotes the message queue size, q the acknowledgment queue size, c and L the capacity and maximum latency of the message channel, and c_{ack} and L_{ack} the capacity and maximum latency of the acknowledgment channel.

receiver of the system is the source, and the sender the sink of information. It is obvious that the whole system stabilizes after the receiver reaches a correct state since the sender overwrites its value of the window size every time it receives an acknowledgment. Even if acks are dropped at the sender's queue, the protocol works since every ack contains the correct window size.

After the correct window size is established, no messages or acknowledgments are lost. The window washing protocol begins to stabilize and eventually the whole system becomes stable.

We will see that the entire protocol reaches a valid state approximately after $3\Delta T$, where ΔT denotes the timeout period for resending messages and acknowledgments. Its value must be greater than the longest possible time between sending a message and receiving the corresponding acknowledgment in the error-free case.

It is assumed that a sender regularly sends messages to the receiver. A state in which one of the senders no longer sends messages is regarded as an erroneous state. This is required by both, the window washing protocol and the window size assignment protocol.

Let us assume the system starts from a random state. The receiver starts to send out acknowledgments to the sender. Before doing so, the receiver checks its window size settings. If the value does not conform to the requirements ($w_{min} \leq w \leq w_{max}$), it will be set to w_{min}. The receiver side of the window size assignment protocol is then in a valid state.

After reading the acknowledgment, the sender overwrites its own window size value with the new value carried in the ack. Now the window size assignment protocol of the sender is in a valid state, too.

The size of the acknowledgment queue of the sender is denoted as q, the maximum channel latency as L_{ack}, and the minimum service rate of the acknowledgment queue as λ.

Latest, after ΔT time units the receiver sends an acknowledgment which piggybacks the correct window size. It is guaranteed that every ack carries a valid window size by calculating w before sending the ack. The ack reaches the sender queue at $\Delta T + L_{ack}$. However, it may happen that the receiver sends the ack earlier and the ack is dropped. By time $L_{ack} + 1/\lambda$ the sender frees a space in its queue. Adding both times results in $\Delta T + 1/\lambda + 2L_{ack}$ for the latest time when an ack reaches the sender. After another $1/\lambda$ (the arriving ack sees an empty queue because $q/\lambda < \Delta T$) the ack is taken out and the correct window size is established. The worst-case adjustment time adds up to $\Delta T + 2/\lambda + 2L_{ack}$.

After the window size is adjusted, no messages are lost due to buffer overflow. Window washing is guaranteed to stabilize within two round trip delays (proof in [2]), which is less than $2\Delta T$. The total time for self-stabilization is smaller than $T_{stabilize} = 3\Delta T + 2/\lambda + 2L_{ack}$.

4 Many-to-One Protocol

Having shown that our protocol stabilizes within $T_{stabilize}$ we now expand the model as it is depicted in Figure 5. Many senders feed messages to a single receiver. All messages share the same queue of size Q. Every sender incorporates an acknowledgment queue of size q.

4.1 Description of the Protocol

On the sender's side only minor changes need to be made. In addition to the lower window edge L_i, the window size w_i and the sequence number seq_i, messages are tagged with the sender number i. Apart from this modification all senders run the one-to-one protocol.

To achieve self-stabilization, several variables are needed on the receiver's side. The internal structure of the receiver is given in Figure 6. The window-assignment protocol uses three arrays: *sender*, *actual*, and w. The first data structure is a shift register that stores the sender ids of the messages received. The capacity of the register is Q, which is also the number of messages the receive queue can hold.

Every time a new message is removed from the receiver queue, the sender id is stripped off and put in the shift register. The protocol then counts the number of identifiers a sender i has in the register and stores the result in $actual[i]$. The *actual* array contains the latest statistics about the traffic distribution. Based on this information, the new window sizes are calculated, stored in the window size array w, and sent to the senders by piggybacking them to the acknowledgment.

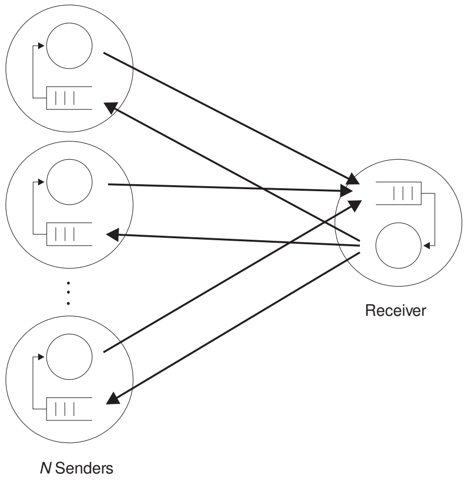

Fig. 5. Many-to-one setup. N senders send messages to a single receiver. All messages are queued in a queue of size Q.

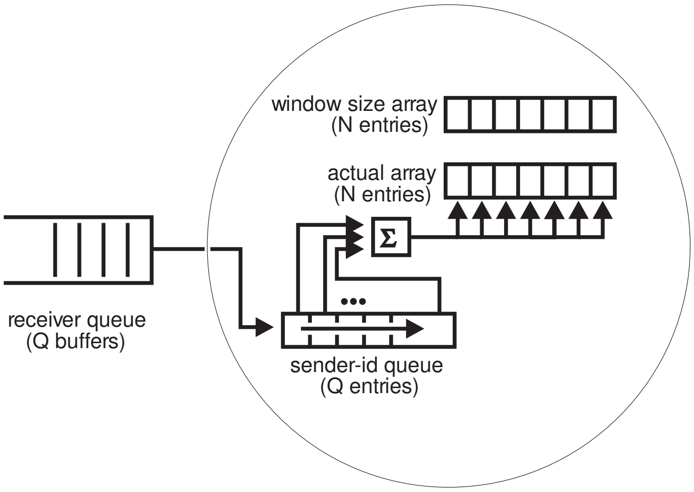

Fig. 6. Internal structure of the receiver. It consists of a message queue and the servicing node. The node uses the internal arrays *sender*, which stores the sender identifiers of the last Q messages that were taken out of the queue, *actual*, which stores the distribution of senders for the last Q messages, and w, which holds the assigned window sizes for each of the senders.

It is easy to see that there is no conflict between the senders if the window size assignment follows several rules. As in the one-to-one protocol, the window size can be increased in arbitrary steps without informing the corresponding sender. However, the receiver can only reduce a window size of a sender that is going to receive an acknowledgment, and the reduction must not be larger than the number of messages that are acknowledged with the ack. The reasoning is identical to that used for the one-to-one protocol.

Assigning new window sizes happens every time a message is taken out of the receiver queue. First, it is determined whether or not the sender j whose message was taken out can be deprived of one unit of its window size. Therefore, the receiver checks if j's margin, which is defined as $actual[j] - w[j]$, is larger than a threshold value $DecThreshold$. If this is the case, the receiver picks the sender k that has the smallest window of the senders with the smallest margin. If its margin is smaller than $IncThreshold$, j's window is reduced by one and k's window increased by one.

The protocol takes a while to adjust the window sizes correctly in case one sender sends heavily and suddenly another sender sends a burst of messages. To increase the dynamics of the system, a less stringent version was developed that allows the receiver to deprive j of one unit of its window even if its margin is less than or equal to $DecThreshold$, as long as it is greater than $margin[k] + 1$.

Pseudo-code for the protocol is given in Figures 7 and 8. Figure 7 describes the send, receive, and check routines, whereas the window size assignment protocol is given in Figure 8. The latter also contains the main functions for both sender and receiver.

4.2 Proof of Self-Stabilization

In the protocol described here several senders write to the same receiver. This does not create an entirely new situation different from that described for the one-to-one protocol. The receiver can be thought of as being split up into N receivers, each of them forming a sender-receiver pair with its corresponding sender. That all of the messages go to the same receiver input queue does not affect the window washing protocol so long as there is no buffer overflow. Therefore, the sum of all of the assigned window sizes must not be greater than the queue size Q.

The window size assignment protocol works exactly the same way as in the one-to-one protocol. Thus, after $\Delta T + 2/\lambda + 2L_{ack}$ all senders are supplied with the correct window size and the system stabilizes two round-trip delays later. Again, we have a worst case stabilization time of $T_{stabilize} = 3\Delta T + 2/\lambda + 2L_{ack}$.

With many senders, the check for a correct window size assignment at the receiver differs slightly from the one-to-one protocol (see Figure 7).

```
SendData( L, w, sender, seq, message )        /* Sender emits message */
        Preconditions: seq ∈ [L + 1, L + w]

ReceiveData( L, w, sender, seq, message )     /* Receiver absorbs message */
        if (sender ∈ [1, N])then
            if (R[sender] ∉ [L, L + w] or seq = R[sender] + 1) then
                R[sender] = seq
                deliver message
            endif
        endif

SendAck( j, w_j, ack_j )                      /* Receiver emits ack to sender j */
        Preconditions: j ∈ [1, N]
        CheckWindowArray()

ReceiveAck( w_j, ack )                        /* Sender absorbs ack */
        w = w_j
        if (ack ∈ [L + 1, L + w]) then
            L = ack
        endif

BuildActualArray()                            /* Calculate values of actual */
        for i in 1 to N do actual[i] = 0      /* set actual[] = 0 */
        for i in 1 to N do actual[sender[i]] + + /* count sender-ids */

CheckWindowArray()                            /* Check values in w[] */
        for i in 1 to N do                    /* first, check if all values
                                                 are in the valid range */
            if (w[i] ∉ [w_min, w_max]) then
                w[i] = w_min
            endifenddo
        if (Σw[i] > Q) then                   /* Check if sum of window */
            for i in 1 to N do w[i] = w_min   /* sizes is smaller than Q */
        endif
```

Fig. 7. Send, receive, and check routines for the many-to-one protocol. Send and receive are similar to the one-to-one protocol.

```
CalculateNewWindow( sender )                   /* Calculates new window size
                                                  for senders */
    for i in 1 to N do                         /* Calculate new margins */
        margin[i] = w[i] - actual[i]
    enddo
    if (∃(margin[] < IncThreshold)) then       /* Find sender whose window
                                                  will be increased */
        k = sender with smallest w of senders with minimum margin
    endif
    if (Σw[i] < Q) then                        /* Free buffer space available */
        if (k exists) then w[k]++
    else                                       /* No free buffer space available */
        if (k exists AND margin[sender] > DecThreshold) then
            w[k]++
            w[sender]--
        endif
    endif

Sender:
    if timeout then w = w_min endif
    if not (w_min ≤ w ≤ w_max) then w = w_min endif
    SendData(L, w, sender, seq, message)
    ReceiveAck(w, ack)

Receiver:
    ReceiveData(L, w, sender, seq, message)
    EnqueueSender(sender)
    BuildActualArray()
    CalculateNewWindow(sender)
    SendAck(sender, w[sender], R[sender])
```

Fig. 8. Many-to-one protocol.

5 Simulation

The many-to-one protocol was evaluated using a discrete event simulation. The simulation model was designed to match the system model given in Section 4. The channels were designed as FIFO queues of length one.

5.1 Error-Free Simulation

For the error-free case, the simulation was started from a quasi-random state. Every sender started creating messages either in a deterministic manner or randomly with a negative exponential distribution. The rate was set to $\lambda_0 = 0.04$. The receiver removed messages from the queue at a constant rate $\lambda_r = 1$. A newly created message was sent if the flow control allowed it; otherwise, it was delayed. The measurements were taken after reaching a state of equilibrium. To measure the ability to adapt to changes in traffic, bursts were injected for senders one and two. The burst rate was $\lambda_b = 0.4$. The results are shown in Figure 9.

From the figures it can be seen that the first window assignment method is superior to the less stringent method in reducing fluctuations of the window size during periods of no change in the sender rates. However, the second method proves to be quicker in redistributing the resources after sudden shifts in the sender rates. This produces a shorter latency for messages and makes the protocol better suited to a changing environment. For these reasons the less stringent window allocation protocol was chosen as the basis for further tests.

In all three diagrams the black sender sends bursts from time 2000 to 3000. The moment the black sender stops, the grey starts bursting for 1500 units. It takes over the black sender's share of the receiver buffer. At time 4000 the black sender again starts bursting, and the shares average out. At 5000 no sender is bursting, and the buffer is equally shared between all five senders.

5.2 Error Recovery

To test the error-recovery times of the protocol, three different types of errors were introduced into the system: loss of messages and acknowledgments, corruption of sequence numbers of messages and acknowledgments, and berserk senders. The statistics used to measure the error-recovery time in these cases were the retransmission count of the sender and the received count of the receiver. When the protocol is in the stable state, both of these measures should be equal to one for every message sent.

The simulation results show that a loss of a single message does not affect the protocol beyond the loss of that message. If the missing message were to be retransmitted, only a small penalty would be added to the recovery process. The same behavior occurs for the second type of error. The receiver accepts the incorrect message and sends an acknowledgment to the sender. Since the sequence number on the ack is invalid, the sender disregards it and continues as if the error had not occurred. These two errors are thus indistinguishable to the senders.

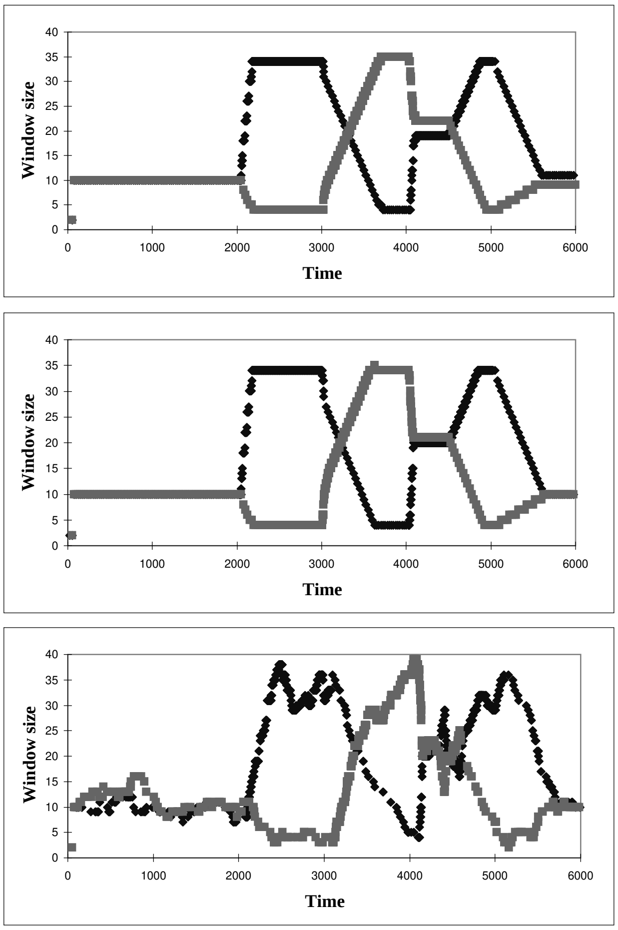

Fig. 9. Simulation of the error-free case: The figures show two senders competing for buffer space. They represent the change in the sender's window sizes over time. The top one depicts the results achieved with the more stringent version ($DecThreshold = 1$), the middle and lower ones use the less stringent version with $DecThreshold = margin[k] + 1$. The IncThreshold equals one for all cases. The upper two graphs illustrate the behavior for deterministic traffic, the lower one deals with a negative exponential distribution of traffic.

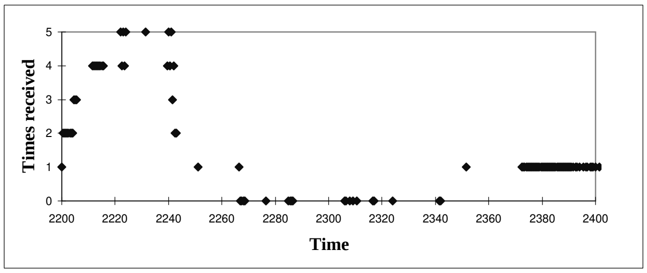

Fig. 10. Simulation of a berserk sender scenario. In a system of five senders, one starts to emit messages at a high rate ignoring flow control for a certain time interval. In this case the interval is set from 2200 to 2300 time units. The figure shows the number of times the same message is received versus the time it is generated. After 180 time units the protocol stabilizes and the system operates correctly.

The third error shows the resilience of the protocol. A sender emits messages at a high rate, ignoring flow control for a time period of 100 time units. From Figure 10 it can be seen how the protocol stabilizes once there are no more new errors. The error-recovery time is shown to be on the order of 200 units.

5.3 Stabilization from a Random State

To test the self-stabilizing bound of the protocol, the system was started with the system in an uninitialized state. The variables were not initialized and random messages were placed in the channels and queues. The system consisted of 10 senders and a queue size of 50 and 4000 simulation runs were executed.

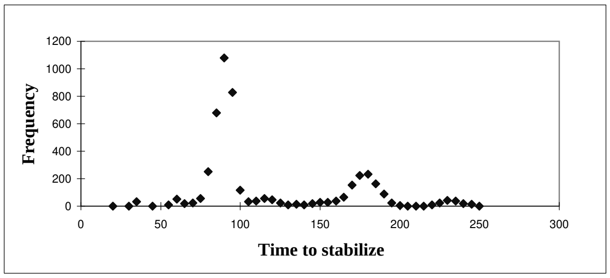

Fig. 11. Simulation of stabilizing time from a random state. In a system of ten senders, the system is started from an uninitialized state. The simulation is run 4000 times. The figure shows the frequency with which a stabilizing time occurs.

The statistics used to measure the stabilizing time were the same as in the error-recovery cases. The theoretical upper bound on the stabilization time was calculated to be 300 time units. The simulation results are shown in Figure 11.

6 Conclusion

In this paper we introduced a new sliding-window protocol that provides reliable communication and flow-control for many-to-one setups. The two main features of the protocol are a flexible way of adjusting the size of the windows to different traffic patterns and self-stabilization. A proof and an upper-time bound are given.

The adjustment of the window size should depend heavily on the statistical properties of the traffic. Constant bit rate traffic and long bursts should not challenge the protocol, whereas short bursts and long pauses are more difficult to handle.

As further work, it would be interesting to see how the protocol behaves when channel latencies are introduced, especially when they vary between senders. This would provide results closer to the real-world scenarios we face. Another interesting line of research would be to examine how this protocol could be merged with a self-stabilizing one-sender/multiple-receiver protocol. The properties of the resulting multiple-sender/multiple-receiver protocol would be of great interest to the distributed computing community.

References

1. G. M. Brown, M. G. Gouda, C. Wu, "Token systems that self-stabilize," *IEEE Transactions on Computers*, vol. 38, no. 6 (June 1989), pp. 845–852.
2. A. M. Costello, G. Varghese, "Self-stabilization by window washing," *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, USA (May 1996), pp. 35–44.
3. E. W. Dijkstra, "Self stabilization in spite of distributed control," *Communications of the ACM*, vol. 17 (1974), pp. 643–644.
4. M. G. Gouda, N. J. Multari, "Stabilizing communication protocols," *Technical Report TR-90-20*, Dept. of Computer Science, Univ. of Texas, Austin, (June 1990).
5. S. Katz, K. Perry, "Self-stabilizing extensions for message-passing systems," *Distributed Computing*, vol. 7, no. 1 (August 1990), pp. 17–26.
6. M. Schneider, "Self-stabilization," *ACM Computing Surveys*, vol. 25, no. 1 (March 1993), pp. 45–67.
7. J. M. Spinelli, "Self-stabilizing ARQ protocols on channels with bounded memory or bounded delay,", *Proceedings of the 12th Conference of the IEEE Computer and Communications Societies*, San Francisco, CA, USA (March 1993), pp. 1014–1022.
8. A. S. Tanenbaum: *Computer Networks*, third edition, Prentice-Hall, Englewood Cliffs, N.J., (1996).