Distrib. Syst. Engng **4** (1997) 109–128. Printed in the UK PII: S0967-1846(97)82910-5

High-performance asynchronous atomic broadcast

Flaviu Cristian†, Shivakant Mishra‡ and Guillermo Alvarez§

Department of Computer Science and Engineering, University of California, San Diego, La Jolla, CA 92093-0114, USA

Received 4 May 1996, in final form 8 February 1997

Abstract. We describe two families of asynchronous atomic broadcast protocols that provide good delivery and stability times, use a small number of messages to accomplish a broadcast, distribute the load of ordering messages evenly among group members, use efficient flow-control techniques, and provide gracefully degraded performance in the presence of communication failures. The *pinwheel* protocols are designed for applications characterized by a uniform message arrival pattern. The *on-demand* protocol is designed for non-uniform, bursty message arrivals. The protocols tolerate omission/performance communication failures and crash/performance process failures. Simulation studies for the pinwheel protocols demonstrate that they have superior performance over other well known atomic broadcast protocols, for uniform message arrival rates. We also report measurements taken on prototype implementations, that show very good results and a substantial performance advantage of the on-demand protocol for non-uniform and intermediate message arrival patterns.

1. Introduction

As stronger dependability requirements are placed on computer systems, fault-tolerant distributed systems are becoming increasingly important. A common technique for ensuring continuous availability of critical services is to implement them as a group of servers running on separate machines. The use of server groups creates a need for a *group communication service*, to be used by any server to disseminate *service state updates* to all members of the group. The *atomic broadcast* group communication service ensures that updates from different machines are delivered to all group members *in the same order* despite random communication delays and failures. This guarantee simplifies the development of applications that must maintain consistent replicas of the group state: if the local states of all members are the same when a group is created, and each member applies the same updates in the same order to its local replica, then each member will pass through the same sequence of states. Since atomic broadcast relieves application programmers from the burden of dealing with the difficult issue of maintaining replica state consistency, it is a key service for implementing fault-tolerant distributed systems.

The Team project at the University of California, San Diego aims at designing a set of fault-tolerant services to support the development of dependable distributed systems. Services of interest include time services [13], atomic broadcast and membership services [1, 16, 18], and availability management services [15].

This paper describes the asynchronous atomic broadcast protocols designed for the Team project. Before deciding what atomic broadcast protocols to implement, we evaluated comparatively in [12] several protocols that seemed to have potentially interesting performance characteristics [4, 23, 8, 3]. The main conclusion of our comparison study was that there was no overall best protocol that provided good performance under all operating conditions. Therefore, one of our aims in designing the protocols presented in this paper is to obtain good performance under several different operating conditions. Another observation that we made was that all atomic broadcast protocols compared were designed for applications in which every group member receives updates from its local clients at approximately the same rate—the case of *uniform* update arrival rates. The *pinwheel* family of protocols presented in this paper was designed for applications characterized by a uniform update arrival pattern.

The other protocol of this paper, the *on-demand* protocol, is to our knowledge the first published solution for applications characterized by *non-uniform*, bursty update arrival patterns. For example, video conferencing systems and primary-backup arrangements exhibit traffic patterns in which group members generate (possibly long) bursts of updates in a short period of time, and bursts from different members seldom overlap in time. The atomic broadcast protocols analysed in [12] are not likely to perform well for such cases.

† http://www-cse.ucsd.edu/users/flaviu

‡ Now with the Department of Computer Science, University of Wyoming, PO Box 3682, Laramie, WY 82071-3682, USA. http://www.cs.uwyo.edu/~mishra

§ http://www-cse.ucsd.edu/users/galvarez

0967-1846/97/020109+20$19.50 © 1997 The British Computer Society, The Institution of Electrical Engineers and IOP Publishing Ltd

Our protocols provide good overall performance in the presence as well as the absence of communication failures, and under low as well as high update arrival rates. We have simulated two of the pinwheel protocols along with the protocols described in [6, 23, 8, 3] under uniform computing conditions. The simulation results show that the pinwheel protocols have superior performance compared with the other protocols for uniform update arrival rates. We have also implemented the pinwheel and on-demand protocols on a network of Sun IPX SPARCstations connected by an Ethernet. Our measurement results show that, even though the pinwheel protocols are better for uniform arrival rates, the on-demand protocol outperforms them for bursty arrivals and for most intermediate scenarios.

This paper is organized as follows. In section 2 we describe the motivation behind our protocols. In section 3 we describe the system model and failure assumptions, as well as the properties of the underlying group membership service assumed by our protocols. Section 4 describes the design and implementation of the pinwheel protocol family. A performance comparison between these protocols and the protocols of [6, 23, 8, 3] is also included in this section. Section 5 describes the design and implementation of the on-demand protocol. A performance comparison of this protocol with the basic pinwheel protocol is also included in this section, as well as qualitative comparisons with previously published protocols. Section 6 discusses the significant performance improvements achieved by the on-demand protocol by batching more than one update in each network message. Finally, section 7 summarizes our findings.

2. Motivation

We consider the following performance metrics in our evaluation of protocols: average broadcast delivery time, average broadcast stability time, average number of messages needed to broadcast an update, throughput, and distribution of processing load among group members. The broadcast *delivery time* is defined as the duration between the moment an update is entrusted to the broadcast service by a group member and the moment the update is delivered by the service to every group member. When it is possible for a message to get lost, the broadcast servers that implement the distributed broadcast service must store each update in a local buffer until they learn that the update is *stable*, i.e. received by all group members. The broadcast *stability time* is the duration between the moment a broadcast server receives an update to be broadcast and the moment all broadcast servers learn that the update is stable. Stability is important for two reasons: first, it is a measure of how long a message stays in the buffers of group members and how much buffer space a protocol needs; second, stability time and delivery time are the same for atomic broadcast services that achieve *strict agreement* [8], i.e. agreement on a unique broadcast history despite any number of processor or communication failures. The *number of messages* needed to broadcast an update includes all messages sent by different group members to complete the broadcast of the update. If the underlying communication network is a broadcast channel, the dissemination of an update requires a minimum of one message. In a point-to-point network with N nodes, an update dissemination requires a minimum of $N - 1$ messages. The *throughput* of a broadcast protocol is defined as the number of updates delivered per second at each node for a given update arrival rate. The *distribution of processing load* among group members refers to how busy a member is in relation to other members, over some sufficiently long period of time (so that all group members initiate about the same number of broadcasts during that period). We approximate this metric by the number of messages processed by a group member over this period.

The protocols to be presented are a consequence of the comparative study [12] of several sequencer-based and token-based protocols that seemed to provide the best compromise between simplicity and performance [4, 23, 8, 3]. In a sequencer-based protocol, a unique broadcast server, the *sequencer*, imposes a total ordering on all updates originating from all broadcast servers. Furthermore, there are fixed-sequencer-based protocols (such as [4, 23, 3]) in which the sequencer is always the same node in the absence of crashes, and rotating-sequencer-based protocols (such as [6]) in which the role of sequencer is taken in turn by all group members. In token-based protocols, a token circulates among all broadcast servers following a fixed cyclic order. A member can broadcast updates only when it holds the token, which explicitly or implicitly carries sequencing information. The protocols proposed in [8, 2] belong to this class. Finally, in communication-history-based protocols, the ordering information is inferred independently by each group member based on a fragment of communication history encoded in each protocol message [25, 24]. We limited our comparative study to sequencer- and token-based protocols, because history-based protocols can require a rather large number of messages to broadcast an update when only one (active) member wishes to broadcast, and can be quite demanding on computational and storage resources. The results of our comparison for the case of uniform update arrival rates can be summarized as follows:

- Broadcast delivery time in the absence of communication failures is much better for sequencer-based protocols than for token-based protocols, particularly under light loads.
- Broadcast stability time for token-based protocols is better than for most sequencer-based protocols (which make extensive use of negative acknowledgments to save on message traffic), particularly when the update interarrival times are large.
- While for a sequencer-based protocol the use of positive acknowledgments improves stability time, it also increases the number of messages sent per broadcast update.
- Protocols based on token circulation or on positive acknowledgments perform better than sequencer-based protocols (that rely on negative acknowledgments) in the presence of communication failures. For the latter protocols, delivery and stability times degrade significantly in these failure scenarios.
- Simple-minded flow control techniques, such as 'at most one outstanding broadcast per group member'

[23, 4, 6], degrade performance at high update arrival rates or when communication failures occur.

• In fixed-sequencer-based protocols, the sequencer is overloaded; in token-based protocols, approximately the same load is evenly distributed among all group members.

Whilst arriving at these conclusions for uniform update arrivals, we also concluded that non-uniform, bursty update arrival rates are not well supported by existing broadcast protocols. In fixed-sequencer-based protocols, the sequencer will seldom be the node that is broadcasting a burst of updates. The sequencer has to take part in the dissemination of every update even if it never initiates broadcasts, thus becoming a bottleneck and causing an uneven distribution of the processing load. Also, the overhead of sending a message from the initiator to the sequencer and another message from the sequencer to all the nodes is excessive. Token-based protocols are better suited for the problem under consideration. However, since the cyclic node ordering is fixed for a given group of active nodes, the token circulates among group members regardless of whether they want to broadcast or not. This leads to long waits to acquire the token, while it is circulating among nodes that have no need for it. Existing token-based protocols give every group member a fair opportunity to broadcast, blocking a node during whole token rotations to let it broadcast a long burst, or learn about the stability of already-sent updates. A similar objection applies to rotating-sequencer-based protocols.

One of our aims was to design atomic broadcast protocols that provide: (1) delivery times as good as sequencer-based protocols in the absence of communication failures, (2) stability times as good as token-based protocols, (3) distribution of processing load as even as that of token-based protocols, and (4) message complexity as good as negative-acknowledgment-, sequencer-based protocols. Another aim was to investigate novel flow control techniques that ensure good performance at very high update arrival rates. A third aim was to minimize performance degradation in the presence of communication failures, by using design techniques similar to those found in token- and positive-acknowledgment-based protocols. All protocols described in this paper achieve these aims. The pinwheel protocols are better suited for uniform update arrival rates; the on-demand protocol complements them by supporting bursty communication in an efficient way.

3. Assumptions, definitions and requirements

3.1. System model

We assume a timed asynchronous distributed system [18] in which network nodes do not share any storage and communicate only via datagram messages and by measuring the passage of time on their local hardware clocks. Our underlying system model is significantly different from the better known *time-free* asynchronous system model [21] characterized by the following properties: (1) services are time-free: their specification describes what outputs and state transitions should occur in response to inputs without placing any bounds on the time it takes these outputs and state transitions to occur, (2) interprocess communication is reliable: any message sent between two non-crashed processes is eventually delivered to the destination process, (3) processes have crash failure semantics, and (4) processes have no access to hardware clocks. Because in the time-free model an observer cannot distinguish between correct, slow or crashed processes, most of the group communication services that are of importance in practice, such as consensus, election or membership, are not implementable [21, 27, 5].

The *timed* asynchronous system model, introduced without being named in [7], and later named in [18], assumes that (1) all services are timed: their specifications prescribe not only the outputs and state transitions that should occur in response to inputs, but also the time intervals within which a client can expect these outputs and transitions to occur, (2) interprocess communication is done via an unreliable datagram service with omission/performance failure semantics [9]; communication failures are rare events: most messages are *likely*† to reach their destination within a known delay δ, (3) processes have crash/performance failure semantics; process failures are rare events: after starting, processes are likely to be timely, (4) processes have access to hardware clocks that run within a known linear envelope of real-time, and (5) no bound exists on the rate of communication and process failures that can occur in a system. Our use of this model instead of the time-free model is based on our belief that it adequately describes existing distributed systems built from networked workstations, and because (unlike the time-free model) the timed model allows practically needed services such as clock synchronization, membership, consensus, election, and atomic broadcast to be implemented [7, 18, 19, 10, 20]. For a more precise, formal description of the timed asynchronous system model see [14].

3.2. Underlying membership service

The atomic broadcast protocols to be described assume the properties of the three-round majority *group membership* protocol formally specified, described and proven correct in [18]. For brevity, we only mention here what is needed to understand this paper, referring the interested reader to [18, 10] for the formal specification of the assumed membership service.

Given a *team* of processes that collectively implement a certain distributed service, the membership service generates a linear history of completed majority groups of team members. The *membership* of each such majority group and its *unique identifier* are known to, and agreed upon by, all group members when they join the group. A group is *completed* if it is joined by all its members. Completed groups enable members to atomically broadcast service state updates. Broadcasts are not available in (short-lived) incomplete groups, that might be created by the membership service when the failure rate experienced by the system is too high. From an operational point

† For a precise definition of what we mean by 'likely' see the section titled 'Choosing a failure semantics' in [9].

of view, the group membership service interacts with the local broadcast servers via upcalls, notifying them of the formation of new majority groups due to node crashes, joins, and communication failures and recoveries. Since we are only concerned with majority groups in this paper, we simply refer to them as groups in what follows. Sections 4.1.3 and 5.2.3 describe how our protocols react when new groups are formed as a consequence of failures or recoveries. We denote by N the number of members of the current majority group. Since we assume that the identifiers of team processes are totally ordered, it is easy to define for each member of this group a unique *rank* which is an integer between 0 and $N-1$. We refer to the group member with rank i as *group member* i in what follows.

3.3. Atomic broadcast properties

The atomic broadcast protocols presented in this paper achieve *majority agreement*.

Broadcasts can take place only in majority groups. We draw a distinction between a majority group member's proposing an update, and the delivery of that update to each receiver. In the absence of failures, all protocols to be described ensure that all group members deliver all proposed updates in the same order, and that this unique global order preserves the local order in which each sender proposes its own updates. In the presence of membership changes, majority agreement protocols guarantee that whenever a process p delivers a proposed update u, then eventually either all group members deliver u, or a new group g is created that does not contain p as a member such that no member of g has delivered u, and any update proposed by p after delivering u is not delivered by any member of g. Majority agreement also implies that the delivery order is an extension of the causal order (*causal delivery*), provided all interprocess communications are done via the atomic broadcast service.

It is possible to modify our protocols to achieve *strict agreement* [10], i.e. to ensure that any two team members, whether joined to majority or minority groups, agree at any time on a unique history of broadcast updates. Strict agreement can be implemented by making each broadcast server delay the delivery of updates until it learns of their stability. In this paper we concentrate on the majority-agreement versions of the protocols. We refer the interested reader to [10] for the formal specification of atomic broadcast properties.

4. Pinwheel protocols

The pinwheel protocols possess the best qualities of sequencer- and token-based protocols, for uniform update arrival rates. They use the idea of a rotating sequencer to impose a total order on the updates being broadcast, as the best compromise between fixed-sequencer-based and token-passing-based atomic broadcast protocols.

Initially, we describe the *basic pinwheel protocol* to introduce the fundamental ideas of our approach. The *pinwheel protocol with newsmonger* applies a novel technique for reducing delivery and stability times at low update arrival rates and in the presence of communication failures. Although we describe this technique in the context of the pinwheel protocol, it can be applied to many other previously published protocols. The complete pseudocode for the pinwheel protocols can be found in [17].

4.1. Basic pinwheel protocol

When a group member with rank s receives a request to broadcast an update u, it sends a *proposal* message $\langle u, s, n\rangle$ to all other group members. Each proposal is uniquely identified by the identity s of the proposer and a monotonically increasing local sequence number n (initially $n = 0$). We will refer to $\langle s, n\rangle$ as *update identifier* henceforth.

Proposals are totally ordered by group members, who in turn, play the role of order *deciders* (or sequencers [6]). A decider decides the place a proposal $\langle u, s, n\rangle$ will occupy in the history of delivered updates by associating a monotonically increasing *ordinal number* o with update identifier $\langle s, n\rangle$ and broadcasting a *decision* message containing $\langle s, n\rangle$ and o. Group members deliver updates in the order specified by their ordinals.

After group creation, the first group member that becomes a decider is the one with rank 0 when it learns of one or more proposals that need to be ordered. Next, the member ranked 1 becomes a decider when it learns of decision 0, the proposal(s) ordered by decision 0, and some proposal(s) that have not yet been ordered. In general, the kth member to play the role of decider by broadcasting decision k is the member with rank k mod N, after it learns of decisions $0, 1, \ldots, k-1$, all proposals ordered by these decisions, and some proposal(s) that have not yet been ordered (mod denotes the remainder of division by N). A decider relinquishes the role of decider after broadcasting a decision message. Since a group member can play the role of kth decider only after receiving a decision message from the $(k-1)$th decider, for any $k > 0$, there can be *at most one* decider at any point in time. The reason why we call this the *pinwheel*† protocol is because the role of the decider rotates among members at a speed proportional to the intensity of the update arrival rate. When there are only a few arrivals per time unit, the decider role rotates slowly. Rotation speed increases when the rate of arrivals increases and decreases when the rate of arrivals decreases.

Every group member maintains a *proposal buffer*, in which it stores all unstable proposals. Recall that a proposal $\langle u, s, n\rangle$ is unstable at a member r, if r does not know whether $\langle u, s, n\rangle$ has been received by all group members. A negative acknowledgment technique is used to detect communication failures. When an arbitrary group member r receives a new proposal $\langle u, s, n\rangle$ from s, r first stores $\langle u, s, n\rangle$ in its local *proposal buffer*, and then checks to see if it has missed any proposals from s. It detects a proposal $\langle u', s, m\rangle$ for some update u' and sequence number $m < n$ as missing if it has not yet delivered $\langle u', s, m\rangle$, and $\langle u', s, m\rangle$ is not present in r's proposal buffer. If a missing

† Pinwheel: small wheel (shaped like a helix or propeller) with vanes of paper, pinned to a stick so as to revolve in the wind; a whirling firework.

proposal is detected, r sends s a *missing proposal* message (also known as a negative acknowledgment) that includes the update identifiers of all proposals from s missed so far. Since s keeps all proposals that are unstable at s in its local proposal buffer, the proposals missed by r are in this buffer and will be resent by s. To prevent repeated sending of negative acknowledgments, r will not send new missing proposal messages unless it receives a reply to its first missing proposal message or 2δ time units elapse, where δ is a one-way timeout delay.

When a group member d becomes a decider, it creates a list *decidables* of all proposals that it can order. A proposal, $\langle u, s, n\rangle$, can be ordered by d if the following conditions are satisfied: (1) $\langle u, s, n\rangle$ is stored in the d's local proposal buffer, (2) $\langle u, s, n\rangle$ was not ordered by any previous decision message received by d, (3) all previous proposals from s have been received by d. If the list *decidables* is non-empty, d constructs a *decision* message that includes a decision number k which is one greater than the highest decision number d knows about, the list $decidables$, and a range of ordinals, [s_ord, e_ord], where s_ord is one greater than the highest ordinal d has seen so far and $e\_ord = s\_ord + |decidables| - 1$ (where $|decidables|$ denotes the size of the list *decidables*); this range is used by group members to determine the length of the decidables list. The decision message is then sent to all other group members, stored locally in a *decision buffer* and d relinquishes the role of decider.

On receiving a decision message $\langle k, decidables, s\_ord, e\_ord\rangle$, a group member r first stores it locally in its decision buffer and then checks if it has missed any proposal or decision messages. A proposal $\langle u', s, m\rangle$ is determined to be missing if this proposal has not yet been delivered, is not present in the proposal buffer, and there is an update identifier $\langle s, n\rangle$ in the *decidables* list such that $m \leq n$. Similarly, a decision message with decision number l is determined to be missing if $l < k$, l is not present in the decision buffer, and l is greater than the highest decision number h such that all proposals ordered by the hth decision have already been delivered. If a missing proposal is detected, r sends a missing proposal message along the lines described above. If a missing decision is detected, r sends a *missing decision* message that includes r and the set of decision numbers missed so far to group member (k mod N). However, there is a special case in which a missing decision cannot be detected in this way. After sending a decision message, a group member p relinquishes the role of a decider; the next group member, say q, assumes this role only when receiving the decision message. If q does not receive the decision message, the decider role is lost and the protocol blocks. To mask decision message losses, p should resend the decision message if it does not hear from q within $ms + 2\delta$ time units, where ms is the maximum number of time units that can elapse before the decider sends a decision message (see section 4.1.2). If after a maximum number k of attempts p still does not hear from q, it invokes the underlying membership protocol to form a new group. On receiving a missing decision message, a group member resends the requested decisions, if locally present, just like the original decision messages to the requester. Again notice that the group member (k mod N) is guaranteed to have all these missing messages in its proposal or decision buffers, since he cannot have decided that they are stable.

Each member r maintains an integer variable *hdo* (highest delivered ordinal), initialized to -1. Member r delivers an update u proposed in a proposal $\langle u, s, n\rangle$ and ordered by a decision $\langle k, decidables, s\_ord, e\_ord\rangle$, if (1) $\langle u, s, n\rangle$ is in p's local proposal buffer, (2) the kth decision has been received, and (3) $hdo + 1 = s\_ord + i$, where i is the position† of $\langle s, n\rangle$ in the list *decidables*. After delivery of u, r increments hdo. The condition to deliver an update may be satisfied after receiving a proposal (if the decision that orders this proposal has been received earlier), after receiving a decision, or after delivering another update. So a group member checks the delivery condition after each of these events. After delivery of one or more updates, group member r becomes a decider, if (1) all proposals ordered by hth decision have been delivered by r, (2) $r = (h+1) \bmod N$, where h is the highest decision number received.

In order to establish stability of broadcast updates, group members piggyback their local *hdo* value in all messages sent. Each group member s records for each other member r the highest ordinal $hdo(r)$ delivered by r that s knows about. An update ordered by ordinal o is stable at group member s if $o \leq \min\{hdo(r) | 0 \leq r < N\}$. Once an update u becomes stable at r, r removes the proposal containing u from its proposal buffer. When all updates ordered by a decision become stable, r also removes that decision from its decision buffer.

Theorem 1. An update ordered by decision k is stable at a group member s when s receives the decision $k + N - 1$.

Proof. Receiving the $(k + N - 1)$th decision implies that every group member has sent a decision message with a decision number greater than or equal to k. Since a decider sends the nth decision only after delivering all proposals ordered by the $(n - 1)$th decision and delivers immediately the proposals that it orders, receiving the $(k + N - 1)$th decision in turn implies that every group member has delivered all proposals ordered by the kth decision. □

It follows that the decision buffer contains at most $N-1$ entries at any time. Group members may also use this theorem to establish the stability of updates: on receiving decision k, a group member can purge all decision messages with decision numbers less than or equal to $(k - N + 1)$ as well all proposals ordered by these decisions.

4.1.1. Flow control strategies. Despite the above rules for purging the proposal and decision buffers, it is possible that these buffers overflow. This is more likely to happen when updates arrive at a very fast rate or when communication failures occur often. We propose two alternative flow control strategies to handle these situations.

The first strategy, *overflow avoidance*, rules out buffer overflows completely by restricting the maximum number of unordered proposals a group member may have

† Position numbers start with 0.

outstanding at any time. A group member accepts an update from a local client only when the number of locally originated updates that are not yet known to be stable is smaller than some constant *mou* (maximum own unstable). This strategy is conservative in that it assumes that all group members may send *mou* updates simultaneously. In effect, it restricts a very active group member to send at most *mou* proposals even if other members are not sending any proposals and there is sufficient buffer space to store additional proposals.

Theorem 2. Buffer overflow cannot occur at a group member if its proposal buffer can store at least $N * mou$ proposals.

Proof. Recall that only proposals that are not known to be stable need to be stored in the proposal buffer and that a proposal is purged from the proposal buffer as soon as it becomes stable. Since a group member p sends a new proposal only if the number of proposals originated at p that are unstable at p is smaller than *mou*, it follows that an arbitrary group member q has to store in its proposal buffer at any time at most *mou* proposals per group member, thus at most a total of $N * mou$ proposals for *all* group members. □

The above theorem yields an additional rule for update stability determination: if a member q ever has in its local proposal buffer $m > mou$ unstable proposals from a member p, q learns the stability of the oldest $m - mou$ proposals from p. It also follows that at any time, the decision buffer contains decisions that order at most $N * mou$ proposals.

The second strategy, *overflow tolerance*, fulfils the simultaneous aims of reducing the possibility of overflows while posing fewer restrictions on the number of outstanding proposals, and ensuring progress even when buffer overflow occurs. A member will continue to make progress if it does not drop a proposal that it will deliver next and continues to store all those proposal and decision messages that it may be requested by another member to retransmit. These aims are achieved by restricting a group member from initiating new proposals only when buffer space becomes effectively tight at some group member, and ensuring that when buffer overflow occurs at some member q, q will have to drop messages that contain unordered proposals, but not messages that contain ordered proposals or decisions—which are vital for ensuring progress. Since our protocol was designed to tolerate communication omission failures, dropping messages that contain unordered proposals will naturally be tolerated.

To enforce this strategy, each group member q maintains the number of free slots in its proposal buffer and piggybacks this information on each outgoing message. Each member p records for each other member q the number of free slots $fs(q)$ that it knows q has. A decider orders at most mb (maximum batched) messages with a decision message and each member accepts to store at most *moun* (maximum own undelivered) proposals originated locally. A member must be able to store $N - 1$ decision messages and $(N - 1)mb + moun$ proposals. A member rejects a broadcast request from a local client if the number of locally originated proposals that are not deliverable becomes equal to *moun* or if $\min\{fs(i) | 0 \leq i < N\} < lwm$ (low water mark).

With this flow control strategy, on receiving a decision message, a group member p reserves a slot in the proposal buffer for each proposal ordered by this decision but not received yet. When a new proposal $\langle s, n, u \rangle$ is received and the local proposal buffer is full, p performs the following actions. If there is a reserved slot for $\langle s, n, u \rangle$, because a decision message ordering this proposal was received earlier, p stores the proposal in that slot. Otherwise, let m be the highest proposal number among those received from s that have not been ordered. If no such m exists or if $n > m$, p drops the new proposal, otherwise it drops the proposal with proposal number m and replaces it with $\langle s, n, u \rangle$. When p receives a new decision message, for each proposal $\langle s, n \rangle$ ordered by the decision, p performs the following actions. If $\langle s, n \rangle$ has already been received, it does nothing, otherwise, if there is a free slot available in the proposal buffer, it reserves that slot for $\langle s, n \rangle$, otherwise, p chooses a group member v (victim) such that $v \neq p$ and there is at least one proposal received from v that is not yet ordered. Let m be the highest proposal number from v that has not been ordered. Node p drops $\langle v, m \rangle$ and reserves the slot occupied before by $\langle v, m \rangle$ for $\langle s, n \rangle$. Notice that such a victim v will always be available, since p's proposal buffer can store up to $(N-1)mb+moun$ proposals (a maximum of $(N-1)mb$ proposals ordered in $(N-1)$ decision messages and a maximum of *moun* undelivered proposals originated locally). In our implementation, we found this strategy very useful in avoiding network congestion, because it automatically slows down the source of sending messages when the network congestion possibility occurs.

Theorem 3. The overflow tolerance strategy ensures progress despite buffer overflow occurrences (i.e. a group member always has sufficient space to store decision and proposal messages that may be needed to be delivered next, and always stores proposal or decision messages which it may be asked to retransmit).

Proof. In our protocol, a group member r may be asked to retransmit either (i) an ordered proposal (when some group member detects proposal message loss by receiving a decision), or (ii) a proposal initiated by r (when some group member detects proposal message loss by receiving a proposal from r).

From theorem 2, we know that a buffer large enough to store $N - 1$ decisions will never overflow. So, here we concentrate on the overflow of the proposal buffer. From theorem 1, we know that there can be at most $N - 1$ decisions such that the proposals ordered by these decisions need to be stored; all other ordered proposals can be purged. Since a decision orders at most mb proposals, a group member has to store at most $(N - 1) \times mb$ ordered proposals. Furthermore, a group member can initiate at most *moun* proposals that are not yet ordered. Hence, a buffer large enough to store $(N - 1) \times mb + moun$ proposals guarantees that there is sufficient space to store all ordered proposals, in particular the proposals that need

to be delivered next, and all locally originated unordered proposals. Since this strategy never drops an ordered or locally originated unordered proposal, a group member always has all proposal messages which it may be asked to retransmit in its local proposal buffer. □

4.1.2. Improving performance at low update arrival rates. The delivery time of a broadcast update includes the transmission delay from the sender to the current decider and the transmission delay from the decider to all group members; the first component is zero if the sender of the broadcast update is the current decider. This is similar to the delivery time experienced in a sequencer-based protocol. Hence, the delivery time of the pinwheel protocols is comparable with that of sequencer-based protocols. The number of messages sent per update broadcast is also similar to that of sequencer-, negative-acknowledgment-based protocols. Each group member assumes the role of a decider cyclically, thus the extra processing load of ordering updates is distributed evenly among all group members over a sufficiently long period of time. The flow control techniques described ensure that performance does not deteriorate even when updates arrive at a very fast rate and when communication failures are frequent. Since the stability of an update u is established by receiving some updates broadcast after u, the stability time depends on the update arrival rate. As a consequence, the stability time can grow large when new updates arrive slowly. The exclusive use of negative acknowledgments makes the time to detect a message loss also dependent on the update arrival rate: message loss detection latency grows when the updates arrive slowly.

To improve the performance at low arrival rates, after a member d becomes a decider, we require it to send an *empty decision* message with an empty *decidables* list if it does not learn of any new proposal to order for ms (maximum silence) time units. Empty decision messages are processed just like non-empty ones. They help members detect missing proposals and decisions and help establish the stability of the proposals broadcast before the period of silence began. The introduction of empty decision messages when the update arrival rate falls below a certain threshold leads to a slight increase in the average number of messages sent per broadcast. However, such a background message traffic is needed anyway to check for group membership stability, so we do not regard it as a problem.

4.1.3. Group membership changes. We proceed to describe the actions taken in response to membership change notifications from the group membership service. In order to differentiate between messages sent in some previous group and messages sent in the current group, a group identifier is included in every message. The new group (made known to each node by an upcall from the group membership service) can reflect node departures, joins, or a combination of both. If the new group contains only additions to the previous one, the current decider includes the new group identifier in the decision message it sends. The start ordinal in this decision message is the start ordinal of the new group. New members start delivering updates from the start ordinal of the new group. The new group members deliver only those updates that are ordered by decision messages sent with the new group identifier†. To ensure that all new group members start delivering updates from the same ordinal (the start ordinal of the new group), the start ordinal of the new group is included in every decision message sent in the new group for one round, where a round corresponds to the rotation of the decider role among all group members in the new group. In case a new group member misses the first or a first few decision messages, it will detect this when it receives a decision message for the first time and the start ordinal of the decision message and the start ordinal of the new group (included in the decision message) are not the same. As usual, on detecting a missing decision message, the new member sends a retransmission request to the sender of the decision message that caused the message loss detection.

If at least one member of the old group is not present in the new one, the protocol tries to determine whether the decider was removed from the group. This is done by designating member 0 of the new group as a coordinator. Every other member of the group sends to the coordinator the decider's process identifier—to the best of its knowledge. A member repeatedly sends this information to the coordinator at regular time intervals, until it receives a message from the coordinator. If any of the members claim to be the decider, the coordinator communicates this fact to every member using a broadcast, and the protocol operation continues as described for pure joins. Since a decider relinquishes its role when it sends a decision message and the next member assumes the decider role only after receiving this decision message, not more than one member can ever claim to be the decider. If no member claims to be the decider, the coordinator infers that the decider has departed (i.e. has crashed or is unreachable), and initiates a two-round decider recovery protocol. In the first round, each member states the highest ordinal delivered by it and who the current decider is—to the best of its knowledge. As a result of this first round, the coordinator can determine the highest ordinal delivered by some member of the new group. Let p be one of the processes that has delivered the highest ordinal. The coordinator designates p as the new decider. In the second round, the coordinator communicates the identity of the new decider to all group members, and then the protocol continues as described for pure joins.

The new group members should deliver updates only after all updates with ordinals smaller than the start ordinal of the new group are stable among the surviving members of the old group; otherwise a new membership change before one of the older updates becomes stable might violate the causal order guarantee of the majority agreement semantics. Thus, the role of the decider completes one more round only among the surviving members of the old group to ensure the stability of the older updates; the new members of the group refrain from proposing or delivering updates until this round is finished. In general, deciders do not order any updates proposed by processes that do not belong to the current group.

† The membership service initializes their state prior to such message deliveries as described in [11].

4.2. Pinwheel with newsmonger

While the basic pinwheel protocol bounds the possible silent periods by multicasting empty decisions to improve message loss detection and stability at low arrival rates, this variant of the pinwheel protocol relies on a point-to-point *newsmonger* message that from time to time will rotate twice to achieve similar results.

The newsmonger is generated by an agreed-upon group member (the *newsmonger owner*, for instance the member with rank 0), whenever it does not learn of any new proposed update for at least ms (maximum silence) time units. The newsmonger message completes two rounds among members and then stops again at the newsmonger owner. It carries the following information: a mapping $hp : [0, N-1] \rightarrow Integers$ that records, for each group member r, the highest proposal $hp(r)$ known from r, and two integers hd (highest decision number) and *mhdo* (minimum of highest delivered ordinals).

The purpose of the highest proposal information carried by a newsmonger message is to let members detect missing proposals and ask for retransmissions. The newsmonger owner initializes the hp array to the values known when the newsmonger is generated. When the newsmonger arrives at some member p, p increases the value of an entry $hp(s)$ if it knows of a higher proposal number than $hp(s)$ from s. If p detects that the highest proposal from s that it knows about is smaller than $hp(s)$, it asks for a retransmission from s.

The highest decision number information hd carried around by the newsmonger lets members detect missing decision messages and ask for retransmissions. When the newsmonger arrives at some member p such that the highest decision seen so far by p is smaller than hd, p will ask for retransmissions of all the decisions it has missed; otherwise, if p knows of a decision d higher than hd it will substitute d for hd before relaying the newsmonger to its next neighbour. In case p needs to send a retransmission request for a decision message, it sends it to the member who sent that decision message.

The aim of the *mhdo* value carried by the newsmonger is to determine the stability of the last proposals diffused before a period of slow arrivals starts. The newsmonger owner initializes the value of *mhdo* to the minimum of the highest delivered ordinals of all group members as known locally (remember that each member piggybacks its highest delivered ordinal on all messages it sends). Consider now the case when some arbitrary group member p receives the newsmonger in its first round and let $hdo(p)$ be the highest ordinal delivered at p. If $hdo(p) < mhdo$, p substitutes $hdo(p)$ for $mhdo$ before relaying the newsmonger. In this way, after a first round, the $mhdo$ value carried by the newsmonger is equal to $\min\{hdo(p) | p \in [0, N-1]\}$, which is by definition, the highest stable delivered ordinal. Thus, when a member p sees the newsmonger in its second round, it learns that all proposals ordered by ordinals equal or less than the $mhdo$ values carried by the newsmonger are stable and can be purged.

Since the newsmonger is relayed from each node to its successor in the cyclic order, this member of the pinwheel family is better suited for point-to-point communication networks. The newsmonger is generated only when new updates arrive slowly, but as observed earlier, such background message traffic is also needed for checking membership stability. Thus, the newsmonger plays a double role in speeding update stability at low arrival rates and checking membership stability when no other traffic is present. Although superficially the newsmonger resembles a token, it plays a very different role. It is simply used for 'gossip' to improve protocol performance, not for mutual exclusion in real time. Thus, the generation of multiple newsmongers does not affect the correctness of the protocol, while the presence of several tokens would be fatal for a system that depends on tokens to ensure mutual exclusion. To deal with the possibility of newsmonger loss, any new newsmonger is stamped by the newsmonger owner with a monotonically increasing version number. The newsmonger owner will decide that the newsmonger is lost if it does not receive it back within $N \times \delta$ time units after its generation. In such a case, the owner resends a copy of the newsmonger with the version number incremented by one. A group member p does not relay a newsmonger if its version number is smaller than the highest newsmonger version number seen by p so far.

Notice that although we have described the idea of 'newsmonger' in the context of the pinwheel protocols, it can be applied to many existing broadcast protocols to improve their performance when updates arrive at a slow rate or when communication failures occur. Essentially, a newsmonger carries information about all the updates that have been broadcasted, all the updates that have been ordered, and all the updates that have been delivered by different group members. It circulates among group members from time to time, when there is very little protocol activity, for two rounds. In the first round, it gathers all the information from group members and, in the second round, it conveys this information to the group members. A group member can detect message losses and stabilize updates from the information carried by the newsmonger.

4.3. Performance

We have implemented the pinwheel protocol on a network of Sun IPX SPARCstations connected by a lightly loaded 10 Mbps Ethernet. This implementation consists of about 2000 lines of C code, and uses the UDP broadcast interface of the Unix operating system (SunOS 4.1.2) and Sun's LWP thread management library.

We have measured the average delivery time, stability time, number of messages exchanged to accomplish a broadcast, and the number of updates delivered per second at each group member (throughput). All these measurements have been taken on a group of three servers running on separate processors and independently generating up to 2000 broadcast requests. Update messages are 20 bytes long, of which 4 bytes are for the update itself; decision messages are 56 bytes long for this group size. Since all performance indexes we have measured depend on the rate at which broadcast requests are generated, we have performed different experiments by varying the interarrival

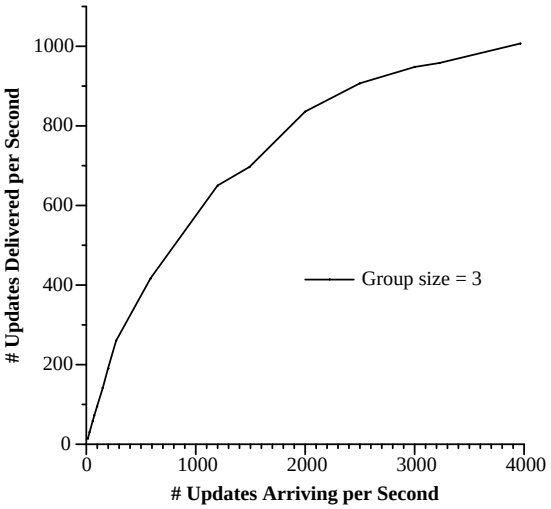

Figure 1. Number of updates delivered per second in the pinwheel protocol.

Table 1. Performance of the pinwheel protocol.

	λ = 200	λ = 500	λ = 1000
Delivery time (ms)	3.0	3.1	3.7
Stability time (ms)	5.2	11.9	15.4
# of messages	1.74	1.39	1.15

time between broadcast requests from 0.75 ms to 400 ms at each participating node.

The performance measurements on this implementation are very encouraging. Figure 1 plots the average throughput as a function of the total number of broadcast update requests generated per second (the *injection rate* λ). The throughput grows approximately at the same rate as λ until a rate of 450 updates per second. After that point, the effect of processing and queuing delays causes the number of updates delivered per second to be smaller than λ. The maximum number of updates the protocol could deliver per second is 1007. This occurs for an injection rate of about 4000 updates per second. Beyond this point, the UDP message loss rate dominates and is the main reason why throughput stops increasing.

The average delivery time, stability time, and the number of messages exchanged to accomplish a broadcast are shown in table 1 for injection rates of 200, 500, and 1000 updates per second. Throughout this paper, the times are indicated in milliseconds.

4.4. Comparison with existing protocols

We proceed to make qualitative and quantitative comparisons between the pinwheel protocols and several existing sequencer-based and token-based atomic broadcast protocols. Because of the similarity between the basic pinwheel protocol and the CM protocol we have also simulated the CM protocol under identical conditions. The good overall performance of the pinwheel protocols comes from several techniques, some of which have been used in other atomic broadcast protocols. These include:

(1) the existence of a single group member ordering many updates simultaneously ensures fast delivery times in the absence of communication failures;

(2) the rotation of the role of sequencer among group members ensures equal processing load over a sufficiently long period of time and fast stability times in the absence of communication failures;

(3) the use of negative acknowledgment techniques to detect communication failures ensures that few messages are exchanged per update broadcast;

(4) the ideas of bounded silence and newsmonger ensure fast stability times and fast message loss detection when updates arrive at a slow rate; and finally

(5) the use of novel flow control techniques ensures progress and good overall performance when updates arrive at a very fast rate or when the frequency of communication failures increases.

Regarding *qualitative comparison*, we will compare the pinwheel protocols with three atomic broadcast protocols proposed earlier. These are the Isis ABCAST protocol [3], the Totem atomic broadcast protocol [2], and the CM protocol [6].

The Isis ABCAST protocol [3] is a fixed-sequencer-based protocol, and so, the following comparison will apply to other fixed-sequencer-based protocols [23, 4] as well. We observed in [12] that the delivery time in the absence of communications failures is best for the fixed-sequencer-based protocols. However, these protocols suffer from unbalanced processing load distribution, large delivery times in the presence of communication failures, and large stability times. The delivery time provided by the pinwheel protocols in the absence of communication failures is as low as that provided by the Isis ABCAST protocol. This is because the mechanism used to disseminate the updates and establish an order of delivery is similar to that in the Isis ABCAST protocol. Unlike the Isis ABCAST protocol, the pinwheel protocols rotate the role of decider to ensure a balanced load and small stability time in the absence of communication failures. In addition, the idea of bounded silence and newsmonger, and the flow control techniques further improve the delivery and stability times over the Isis ABCAST protocol when updates arrive at a very fast rate or when the frequency of communication failures increases.

The Totem atomic broadcast protocol [2] is token-based; the following remarks hold for other token-based protocols such as [8] as well. We observed in [12] that token-based protocols achieve balanced load distribution and low stability times when updates arrive at a fast rate in the absence of communication failures. However, these protocols suffer from long stability and delivery times when updates arrive at a slow rate or when communication failures occur. The stability time when updates arrive at a fast rate in the pinwheel protocols is as low as that in the Totem protocol. This is because when updates arrive at a fast rate, the stability of updates is established by the same mechanism in the Totem protocol as in the pinwheel protocols. Furthermore, like the Totem protocol, the pinwheel protocols result in an even distribution of processing load among all group members. However, unlike the Totem protocol, a group member must not

wait for a token to broadcast an update in the pinwheel protocols. This results in lower delivery times, and lower stability times to some extent, for the pinwheel protocols. In addition, when updates arrive at a slow rate or when communication failures occur, the use of bounded silence and newsmonger in the pinwheel protocols results in a considerably better performance (delivery and stability times) compared with the Totem protocol under similar conditions.

Although the basic pinwheel protocol is similar to the CM protocol, there are several significant differences that result in a better performance of the pinwheel protocol. First, the transfer of the role of order decider (called token site in [6]) is done explicitly in the CM protocol by either sending a message or piggybacking transfer information in another message, and by having the new decider send an acknowledgment message to confirm the transfer. In the pinwheel protocols, the decider role transfer occurs without the need to send any extra messages or information. Secondly, in CM, the proposer of an update uses a positive acknowledgment technique to send a message containing the update to a decider: it continues to retransmit this message at regular intervals until it receives a positive acknowledgment from the decider. In the pinwheel case, a proposer relies on negative acknowledgments to achieve the same effect. This reduces the number of messages sent per broadcast update. Thirdly, in the CM protocol, a proposer is not allowed to initiate a new broadcast until it receives a positive acknowledgment concerning its 'current' proposal. Our performance comparison study [12] showed that such sender blocking worsens the delivery and stability times considerably when updates arrive at a very fast rate or when communication failures occur. The pinwheel protocols use novel flow control techniques that do not block a sender after initiating a broadcast. This ensures progress and good performance even when updates arrive at a very fast rate or when communication failures occur. Fourthly, in the CM protocol all retransmission requests are served by the member playing the role of the decider, while in the pinwheel protocols, retransmissions are done by specific group members, that are individually addressed in point-to-point retransmission requests. This reduces the load on the group member acting as the decider (which already has additional functions to perform). Also, since the role of decider rotates, it is possible that in CM, a retransmission request can be answered by more than one group member: for example, when a group member has accepted the role of decider and the previous decider does not yet know of this fact. Fifth, to avoid floods of retransmission requests, the pinwheel protocols force members to wait a certain amount of time before requesting new retransmissions. This is very useful at high update arrival rates, where a single message loss followed by the receipt of many more out-of-order messages could lead to network congestion due to retransmission requests. Finally, the newsmonger concept contributes substantially to reducing stability time and the time to detect a message loss, without significantly increasing network load. It is worth noting that we deal explicitly with important practical issues such as determining message stability, purging messages from local buffers and issues of flow control and buffer overflow avoidance and tolerance, while [6] was vague on these issues.

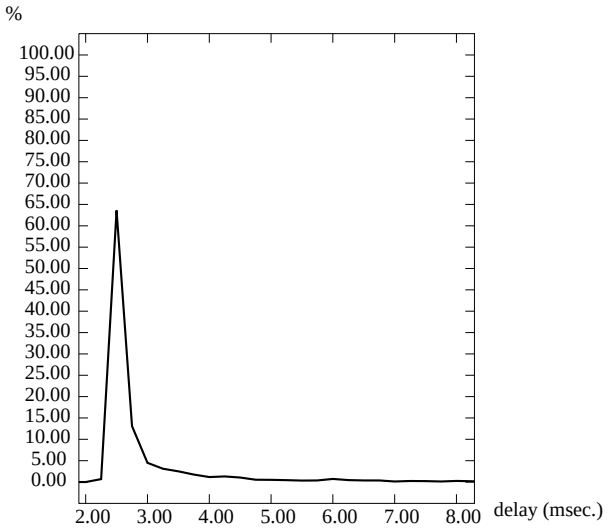

Figure 2. Communication delay density.

For a *quantitative comparison* of the pinwheel protocols, we simulated the behaviour of several sequencer-based and token-based atomic broadcast protocols in a point-to-point network and reported the results of this simulation in [12]. We measured the performance of the protocols on the same computing environment and simulation testbed for a fair comparison. In addition, we also simulated both the pinwheel protocols and the atomic broadcast protocol proposed by Chang and Maxemchuk [6]. The simulation results are very favourable for the pinwheel protocols.

We simulated the sending of 10 000 updates, such that all group members processed the same number of updates in the simulation. The underlying assumptions are described in detail in [12], and can be summarized as follows. The communication delays between group members are assumed to follow the distribution shown in figure 2, which was experimentally obtained; the mean value of the communication delay was 3.318 ms, minimum observed delay was 2.181 ms, and 99% of the observed delays were less than 14.5 ms. The interarrival time between update arrivals at broadcast servers is assumed to be exponential with mean expected value $1/\lambda$. Our simulations have been made for $1/\lambda$ equal to 15, 25, 50, 75, 100, 200, 300, and 400 ms. Finally, we assumed that the CPU time needed to process a message is negligible. (In a real system, this time would amount to tens of microseconds, negligible to the delays of the order of milliseconds caused by the communication network.)

Figures 3 and 4 show the delivery times in the absence of communication failures as a function of group size. We note that the delivery times of the basic pinwheel protocol in the absence of communication failures are similar to those for the sequencer-based Isis [3] and Amoeba [23] atomic broadcast protocols. These delivery times are much smaller than those observed for the token-based Train

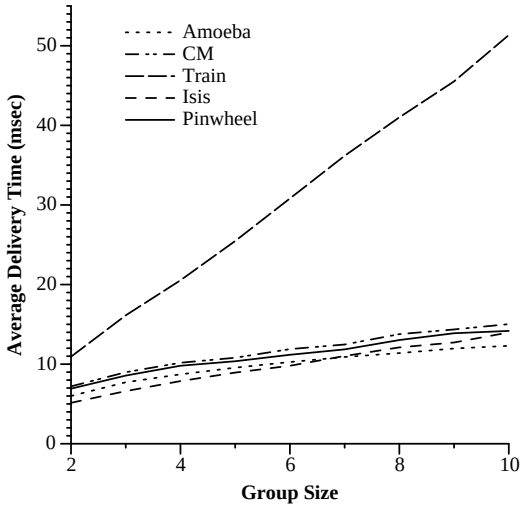

Figure 3. Delivery times ($1/\lambda$ = 15 ms).

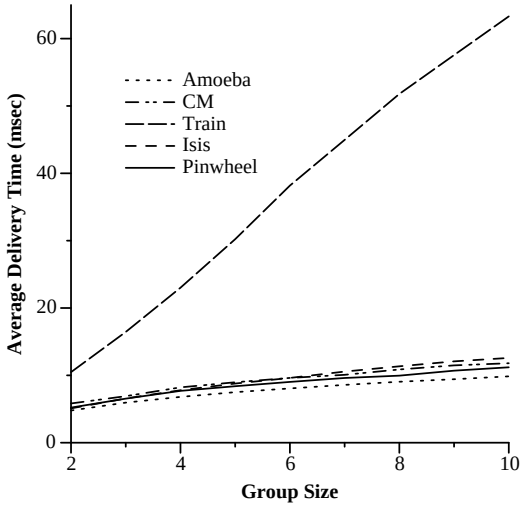

Figure 4. Delivery times ($1/\lambda$ = 300 ms).

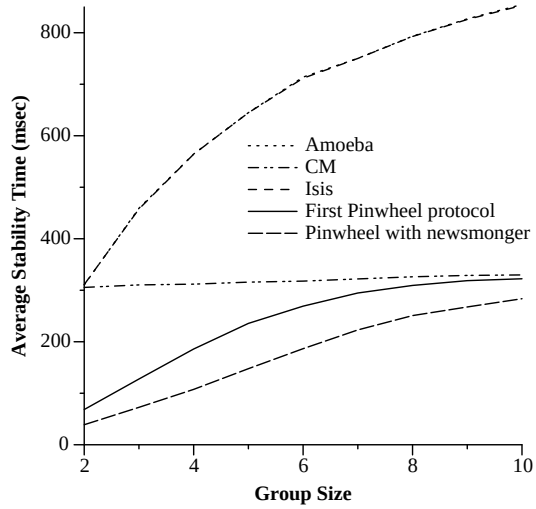

Figure 5. Comparison for $1/\lambda$ = 300 ms.

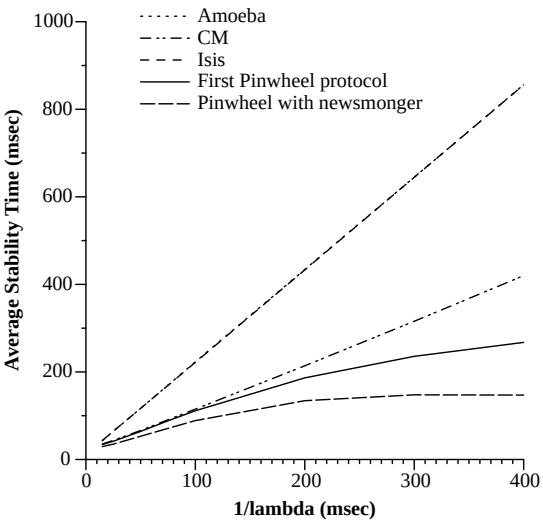

Figure 6. Comparison for group size 5.

protocol. Furthermore, the delivery times of the basic pinwheel protocol in the absence of communication failures are uniformly better than those we observed for the CM protocol; we interpret this as a consequence of the improved flow control techniques.

While the delivery times of the pinwheel protocols in the absence of communication failures are comparable with those of the Isis and Amoeba atomic broadcast protocols, the average stability times of the pinwheel protocols are much smaller than the average stability times of all other protocols we simulated. This is illustrated in figures 5 and 6 (where the curves for Isis and Amoeba overlap).

5. On-demand protocol

The on-demand protocol has been designed to support applications in which each group member generates (possibly long) bursts of updates in a short period of time, and bursts from different nodes are not likely to overlap in time. As an example of bursty update generation, consider a video conferencing application [26], where a set of participants exchange information. In this application, updates are generated from a single group member—the node that holds the floor—most of the time, while the other members remain silent. A similar situation arises in primary/backup arrangements in which the primary is the only sender of updates to a set of backups, but backups can be promoted to play the role of primary because of the occurrence of failures, load balancing reasons, or human intervention. The same traffic pattern can occur when accessing a database partitioned into partitions $P_1, P_2, \ldots$, such that each partition is replicated and is managed by a primary and several backup partition managers. The role of active primary will then migrate among primary partition managers, depending on what partition is accessed at any one time.

As we discussed in section 2, existing atomic broadcast protocols do not support bursty traffic in an efficient way. For the class of applications under consideration, we

claim that there is a tradeoff between protocol performance and fairness. Fast delivery and burst integrity are more important than the ability of each node to broadcast an update at any time. Our aim in designing the on-demand protocol was to achieve the minimum broadcast overhead of a fixed-sequencer-based protocol by imposing that the sequencer and the originator of updates be the same node. For the transfer of the role of sequencer, we propose to move a unique token only among the nodes that have updates to send, and to do that in a fair manner. Only the holder of the token is allowed to broadcast, and it imposes a total order for message delivery that is obeyed by all group members. The protocol exploits the locality principle of the bursty traffic pattern: if a node generates an update, it is very likely that the same node will generate another update soon. The resulting on-demand protocol is intended to have smaller delivery and stability times, and higher throughput than existing broadcast protocols for the bursty traffic pattern under consideration. The protocol processing workload is distributed proportionally to the number of updates originated by each node, and the average number of messages required to propagate an update is very close to one under normal operating conditions.

5.1. Informal description

The on-demand protocol is a moving-sequencer-based atomic broadcast protocol. It uses a token to transfer the role of sequencer among group members, so that at most one sequencer exists at any point in time. We refer to the sequencer node as the *token holder* (or *holder* for short) in the following paragraphs. For the sake of this preliminary discussion, we do not cover the issues of membership changes and communication failure detection and recovery; these points are explained in section 5.2.

As in any sequencer-based protocol, the on-demand protocol requires the holder to assign monotonically increasing sequence numbers (or ordinals) to updates, thereby imposing a total order on their delivery. Since in our protocol only the token holder can broadcast, this node itself assigns the sequence numbers. Thus, the holder determines the ordering of updates as it broadcasts them without any need for additional consensus; it acts effectively as an initiator and sequencer for the duration of the burst.

There are several delicate problems that need to be solved to ensure a fair and efficient operation of the protocol. To become holder, a member must acquire the token. If a potential update sender is not the token holder, it must declare its willingness to broadcast by requesting the token from the holder. The holder will transfer the token to another node only if it has received a token request from that node, thus avoiding the overhead of a fixed cyclic token circulation scheme. We choose to use broadcast messages for the token transfers, so that every group member knows to whom to send token requests. Since the previous holder cannot broadcast more updates after sending the token transfer message, at most a single holder exists at all times.

To guarantee bounded response, the holder must keep the token only for a limited time if somebody else wants to broadcast. Determining when a token holder should release the token has a significant impact on the performance of the protocol. Two conflicting constraints must be considered when defining this policy. First, since the holder is expected to broadcast updates for long periods in bursts, it should be allowed to keep the token for some minimum period of time, even if a request arrives from some other member. Secondly, the holder should not unnecessarily retain the token if it has nothing to broadcast in the near future and it receives a request for the token—we want to keep these idle-token times as low as possible.

The first constraint is easily satisfied by the holder by postponing token transfers until the minimum token holding time elapses. In order to enforce the second constraint, the on-demand protocol uses the following policy. As long as there is no request from another node, the holder keeps the token. When a token request is received, the holder is allowed to continue broadcasting until: (a) the interarrival time between two arriving updates is longer than a certain threshold, or (b) the minimum token holding time elapses from the arrival of the token request—whatever is earlier. By choosing adequate values for the timeout periods the atomic broadcast server can make a good estimation of the end of the burst of updates, while preserving fairness. It is also possible to allow the client applications to make assertions about the burst ends. These assertions are no substitute for the policy described above, as application failures may result in end-of-burst assertions being postponed indefinitely. Instead, assertions should be considered as hints by the broadcast servers.

5.2. Protocol details

We refer the interested reader to [17] for the complete, unambiguous pseudocode of the on-demand protocol. When a group member s receives a request to broadcast an update u, it stores u in a temporary buffer, *updates_to_bcast*, and then checks if it has the token. If s is not the holder and has not requested the token from the current holder yet, it sends a *token_request* message to the holder, and waits for the token to arrive. In the meantime, s stores all broadcast requests received from clients in the *updates_to_bcast* buffer. When the token is present, s drains the *updates_to_bcast* buffer by sending *decision* messages to the non-holders, in the order these updates were received from the clients. The holder sends the updates by using either an underlying broadcast communication mechanism if it exists, or $N-1$ point-to-point messages. Each decision message contains the rank of s, a *group identifier*, the update to be broadcast, and an $ordinal$. The ordinal is a monotonically increasing integer assigned to each update by its originator, and incremented with each update broadcast. The reason for including a group identifier is explained in section 5.2.3. Updates are delivered by group members in the order of the ordinals associated with them.

Every time a non-holder receives a decision message, it stores it in a local *unstable_bcast* buffer and compares its ordinal with the last ordinal received from the holder. If no updates have been lost in the middle, the newly arrived updates are delivered to the client in the correct order. If

there is a gap in the ordinal sequence, the updates will not be delivered until the missing predecessors have been received and delivered.

In general, the broadcast stability time is better when nodes acknowledge the receipt of messages by sending positive acknowledgments [12]. However, this technique increases the number of messages exchanged per broadcast. Furthermore, since we expect potentially long bursts of updates from the holder, it is definitely not practical to require that every member replies with an acknowledgment message every time an update is received. To determine the stability of updates, the on-demand protocol uses a hybrid acknowledgment technique. The whole burst of updates sent by a holder while it keeps the token is divided into *sub-bursts*; non-holders must acknowledge each sub-burst separately. At the end of each sub-burst, an *acknowledgment round* takes place: the holder asks for acknowledgments, the non-holders send them to the holder, and any lost updates are retransmitted by the holder. Each round establishes the stability of ordinals in a given range, hereafter called the *window* of the round. Each acknowledgment contains the updates missed by its sender. Non-holders build *acknowledgment* messages by listing the intervals of missed ordinals; as message losses tend to occur in bursts, this optimization considerably reduces the size of acknowledgments. As in a positive acknowledgment protocol, the sender waits for feedback from the receivers at a globally known point. The acknowledgments themselves are negative: a receiver only requests the ordinals that it did not receive.

To decrease stability times, non-holders include their highest locally delivered ordinal in the token_request message. In this way, the token holder knows which updates are stable and hence purges them from its buffer. Since stable updates must also be purged from the non-holders' buffers, the holder includes the highest stable ordinal in every decision. However, this optimization is not powerful enough to diffuse the highest ordinals in the presence of long bursts: each node requests the token only once, so the holder will not know about updates correctly delivered at each non-holder after it requested the token. We therefore supplement this technique with periodic *max_ordinal* messages sent from the non-holders to the holder, containing the highest locally delivered ordinal at the sender node.

A round is complete when the holder has received acknowledgments from every non-holder, retransmitted missing updates to the nodes that requested them, and received *retransmit_ack* messages back from them. At this point, the holder knows that all updates in the round's window are stable, and purges them from the *unstable_bcast* buffer. If the holder retains the token, it will piggyback the highest stable ordinal in the next decision message; otherwise it will transfer the token to the new holder. Thus, in either case the non-holders learn of the stability of the round's window, and purge the updates from their local buffers as well.

To guarantee that no starvation will occur, a member should receive the token within a bounded period of time after requesting it. For a fair protocol operation, the holder maintains the *requests* queue, containing the nodes that have requested the token. When the holder receives a token_request, it appends the sender's rank at the end of the queue. When the token is transferred, the node at the head of this queue receives it. The queue is included in the *token_transfer* message, so that every member has a knowledge of its contents. The combination of this FIFO management of token requestors and the timeout policies mentioned above implies that a non-holder can wait for the token at most $(N-1) \times$ *max_keep_token* time units, where *max_keep_token* is the maximum amount of time a holder can keep the token when some other node has requested it.

It is easy to see that the processing load is evenly distributed among the group members: the token holder must take care of most of the protocol processing, while non-holders act in a reactive way. In any experiment where each node broadcasts the same number of updates, the number of incoming and outgoing messages will be approximately the same at every site.

5.2.1. Flow control. The length of sub-bursts must be bounded to avoid buffer overflow at the nodes. When the number of unstable updates reaches half the available buffer capacity, the holder requests acknowledgments from all the non-holders by piggybacking the request on the decision. This policy allows the holder to continue broadcasting in parallel while the round proceeds without any noticeable disruption in service. With a suitable selection of parameters, the round will finish before the buffer becomes totally full. Moreover, if the non-holders reply periodically to the holder with max_ordinal messages, and messages are not lost, acknowledgment rounds disappear completely—the window of unacknowledged ordinals never reaches the width that would trigger the initiation of a round, because its starting ordinal moves constantly upwards. If buffers become full, the holder must wait for the ongoing round's completion before starting a new one or broadcasting more updates; rounds are disjoint in time.

This policy may lead to high stability times when the holder is not broadcasting frequently. To solve this problem, we begin to measure two time intervals every time a sub-burst begins or the window of unacknowledged moves upwards. When the first timeout occurs, the holder piggybacks a request for acknowledgments in the next decision regardless of buffer occupancy. If the second timeout occurs without having piggybacked the request (i.e. if no updates were sent after the first timeout) the holder sends a stand-alone *ack_request* message to the group.

In addition to the cases discussed above, a sub-burst can finish for the same reasons that finish the burst when there are pending requests for the token: either the interarrival time between updates is too long, or the token holding time elapses. The maximum number of unacknowledged updates is a configuration parameter of the atomic broadcast service, and should be chosen depending on the quantitative behaviour of the client application. If this number is chosen too large, stability times become excessive; if it is chosen too small, the overhead of acknowledgment rounds dominates. On the other hand, if message losses are infrequent, a larger unacknowledged window results in

amortizing the cost of the round among more user-related messages.

5.2.2. Communication failures. The loss of decision messages is handled as explained above. Missing ordinal gaps are detected by a non-holder in two ways: by receiving a decision with a non-consecutive ordinal, and by receiving a request for acknowledgments (piggybacked or stand-alone) in which the last ordinals of the window have not been received. To recover from the loss of the message indicating the start of an acknowledgment round, the holder waits for 2δ time units. If some acknowledgments are missing at the end of this timeout interval, the holder resends the request to those members who did not reply.

A member detects the loss of its token_request message when the queue contained in the next token message does not contain its own rank. On detecting such a loss, the member resends its token request to the new token holder. Another possibility is that, after requesting the token and waiting for the maximum token holding time, the non-holder does not receive any token transfer. In this case, it resends the token request as well. However, it might be the case that the token transfer indeed occurred, but the requester did not receive it. In this case, its request for the token will arrive at a node that is no longer the holder. Although the requester will notice this mistake sooner or later (i.e. when receiving a decision or request for acknowledgments from the new holder), the old holder forwards the token request to the new holder—to the best of its knowledge. Since this policy could lead to an unbounded wait before acquiring the token in the worst case, only one level of forwarding is implemented.

When a token transfer occurs, the old holder must ensure that the head of the *requests* queue has indeed become the holder, and thus the system is not without a token. Therefore, the old holder waits 2δ for a *token_ack* message from the new holder, and retransmits on timeout if it does not receive it. The old holder does not know if the remaining non-holders have also received the token transfer; but the loss of these messages is not fundamental. Recovery from it has already been discussed in the previous paragraph.

5.2.3. Group membership changes. We proceed to describe the actions taken by the on-demand protocol when notified of the formation of a new group by a membership service that satisfies the properties outlined in section 3. As in the pinwheel protocols, a group identifier is included in every message. If the new group contains only additions to the previous one, the token holder finishes the burst that was in progress when the new group was created, using the old group identifier in the round messages. The newcomers reply with dummy acknowledgments, do not ask for any retransmission, and learn the identity of the holder; the old members participate normally in the round. After the round is finished the holder begins to label its messages with the identifier of the new group and the protocol operation continues.

If at least one node of the old group is not present in the new one, the actions taken by the on-demand protocol are similar to those taken by the pinwheel protocol. To determine if the holder was removed from the group, the member with rank 0 of the new group is designated as the coordinator. Every other member of the group sends to the coordinator the holder's identity—to the best of its knowledge. If any of the processes claims to be the holder, the coordinator communicates this fact to every member using a broadcast, and the protocol operation continues. (Since the token holder relinquishes its role when it sends the token_transfer message for the first time to the new holder, no more than one node can ever claim to be the current holder.) If nobody claimed to be the holder, the coordinator infers that the holder has crashed or is unreachable, and initiates a two-round token recovery protocol. In the first round, each node states which updates it received in the interrupted burst, its intention to get the token, and the last image of the token requesters queue seen by it. As a result of this first round, the coordinator can determine the smallest ordinal not delivered by any member of the group, and gather all the updates sent by the old holder up to that ordinal. In the second round, the coordinator diffuses the updates and the latest version of the queue to all group members. The member placed at the head of the resulting queue is the new holder. Every node delivers the updates diffused by the coordinator before resuming the normal operation.

A combination of crashes and joins for the new group is treated by determining as before if the holder crashed; if it did, newcomers join the operation of the protocol after the two-phase commit. If the holder does not crash, the change is handled as in the case of pure joins.

This recovery policy ensures that the on-demand protocol has total-order delivery and majority agreement semantics. To see why, consider that the holder always delivers its own updates as it sends them to the network. If the holder crashes, the two-phase commit will allow the survivors to deliver only those updates that some survivor has delivered—thus any update delivered only at the holder will not be delivered by any member in the new group. On the other hand, if the holder does not crash, then eventually the acknowledgment round will result in every non-holder's delivering every update in the burst.

5.3. Performance

The prototype implementation of the on-demand protocol runs on the system described in section 4.3, and consists of approximately 3700 lines of C code. The size of the update dissemination messages is 17 bytes: a payload of 4 bytes for the update, plus 13 bytes of control information. (The payload/size ratio can be easily improved by batching many updates in each message as discussed in section 6.) The experiments were performed without batching of updates in either protocol, i.e. by sending a separate update dissemination message for each update. In every experiment, updates were injected at each node at a constant, deterministic arrival rate. All the measurements were taken on a group of $N = 3$ nodes.

The performance measurements were structured as a set of three experiments, with a different traffic pattern

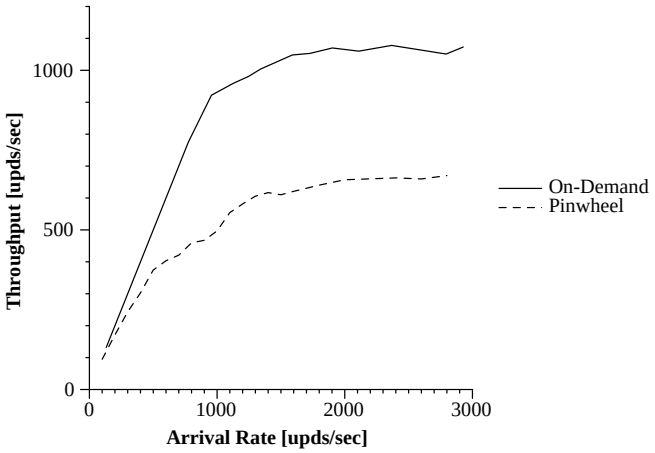

Figure 7. Throughput—one node sends one burst.

Table 2. Delivery, stability, and messages—one node sends one burst.

	On-demand			Pinwheel		
	$\lambda = 200$	$\lambda = 500$	$\lambda = 1000$	$\lambda = 200$	$\lambda = 500$	$\lambda = 1000$
Delivery (ms)	2.11	2.23	2.51	3.25	4.62	7.15
Stability (ms)	5.10	7.22	10.95	6.70	12.40	14.10
#messages	1.11	1.07	1.06	1.97	1.46	1.28

for each of them. We also measured the performance of the pinwheel protocol under the same traffic patterns for comparison purposes. In the first scenario, one of the processors sends a long burst of updates and the others remain silent. This is the experiment that reflects best the bursty traffic pattern for which the on-demand protocol was designed.

Figure 7 shows the plot of throughput (average number of updates delivered per second to each client) as a function of the update arrival rate λ. During transient periods when λ is greater than the throughput, the atomic broadcast service stores the incoming updates in the *unstable_bcast* buffer until the moment in which they can be sent to the other nodes. If the buffer fills up, the client application receives a return value indicating that the broadcast must be retried later—updates accepted from the clients are never discarded.

The interarrival time between updates varies between 0.33 and 10 ms. For the on-demand protocol, the throughput is practically equal to the injection rate until approximately 900 upds s^{-1}. After that point the curve flattens slightly, but it reaches its maximum value of 1070 upds s^{-1} for an arrival rate of 1903 upds s^{-1} and stays close to that value. Beyond 1600 upds s^{-1}, the message loss rate of UDP becomes significant, and the throughput stabilizes at its maximum value. Also, the concurrent use of the main processors to generate the workload contributes to more frequent buffer overflow and lost messages. For the same range of arrival rates, the pinwheel protocol stabilizes at around 660 upds s^{-1} for an arrival rate of 2200 upds s^{-1}, and remains at this level thereafter. The throughput of the on-demand protocol grows much more steeply than the pinwheel's for this case, and the maximum value attainable for the on-demand is also significantly higher. The reason for this is that the on-demand protocol simply sends the burst from the holder in this case, while the pinwheel protocol keeps rotating the role of decider among all the nodes even when the non-holders have nothing to broadcast.

Table 2 shows the average stability and delivery times, and also the average number of messages per update. For each protocol, three representative arrival rates are reported: 200, 500 and 1000 upds s^{-1}. Times are in milliseconds. Delivery times for the on-demand are better than for the pinwheel, and also grow more slowly in this case. The on-demand protocol also has better stability times, that grow with arrival rates because of the increasing message loss probability. The message complexity for the on-demand protocol is close to one for high injection rates, but for lower rates the sub-bursts finish due to a timeout more frequently, thus lowering the number of updates over which each round is amortized. Also, the holder has to send a stand-alone request for acknowledgments more often at low arrival rates. The pinwheel incurs a higher number of messages per update, though smaller than two (one from proposer to decider, one from decider to group members) because the sending node has many updates waiting when it becomes a decider, and they are ordered with a single decision.

The second scenario is the opposite to the first one. Instead of a single node broadcasting freely, all the nodes broadcast a series of updates, and they do it simultaneously. This situation of full competition degrades the performance of the on-demand protocol, as shown in figure 8. In this case the pinwheel has a better throughput than the on-

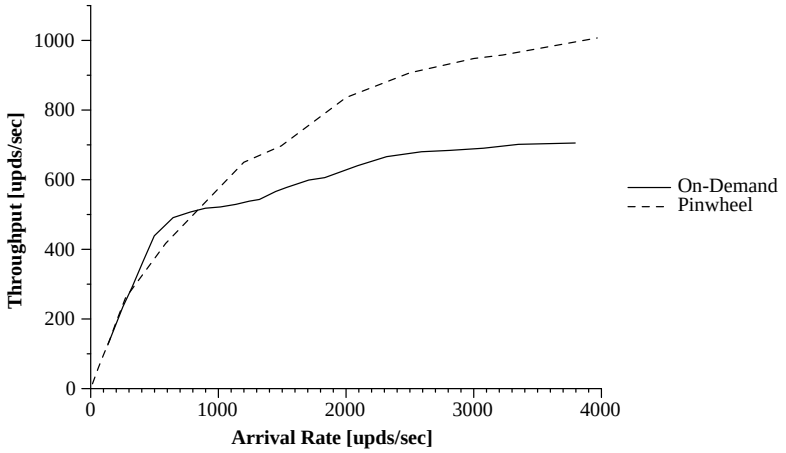

Figure 8. Throughput—full burst overlapping.

Table 3. Delivery, stability, and messages—full burst overlapping.

	On-demand			Pinwheel		
	$\lambda = 200$	$\lambda = 500$	$\lambda = 1000$	$\lambda = 200$	$\lambda = 500$	$\lambda = 1000$
Delivery (ms)	2.85	4.77	9.51	3.0	3.1	3.7
Stability (ms)	6.83	10.1	13.27	5.2	11.9	15.4
#messages	1.46	1.25	1.04	1.74	1.39	1.15

demand protocol; the cause is that the contention for the token forces the holder to finish the bursts prematurely, and transfer the token to another node, which in turn will have to do the same. Therefore, the overhead of rounds is incurred too often, as opposed to the pinwheel in which the rotation of the role of the decider naturally enforces a fair attention of the nodes' requests. However, for the arrival rates smaller than 800 upds s^{-1}, the on-demand protocol has a better throughput, growing again almost as fast as the arrival rate. After that, the overhead of circulating the token becomes excessive and the pinwheel begins to dominate. The maximum throughput attainable by the pinwheel in its ideal case (this scenario) is of approximately 1007 upds s^{-1} for an arrival rate of 4000 upds s^{-1}, while the on-demand protocol reaches 1070 upds s^{-1} for a much lower arrival rate of 1900 upds s^{-1} in the first experiment.

Table 3 contains the performance comparison for the other metrics in the second scenario. The delivery times for the on-demand protocol grow quickly with arrival rates, for the sender no longer has the token at every time. The pinwheel has better delivery times, and is not greatly affected by increasing interarrival rates. The delivery times of the pinwheel improve in this scenario with respect to the previous one. Stability times are also worse for the on-demand, and better for the pinwheel. The message complexity of the on-demand protocol is better than the pinwheel's (although worse than in the first experiment). For this scenario, the pinwheel has a message complexity closer to the intuitive value of 2, particularly for low arrival rates; for higher arrival rates each decision orders increasingly more updates.

The third experiment is an intermediate situation between the stand-alone broadcaster of the first one, and the full contention of the second one. We intend to measure what happens when every node broadcasts a burst of updates, and there is a slight overlapping in time between the bursts. We set this overlapping to be a 30% of the burst length; it is expected to force the holder in the on-demand to release the token before sending all the updates, and thus to be forced to reacquire it later. We show the results in figure 9 and table 4. The throughput for each protocol is between the extreme values of the previous two experiments. The on-demand is here consistently better than the pinwheel for the range of arrival rates tested. Moreover, the on-demand has degraded significantly less than the pinwheel—when the performance in this experiment is compared against the corresponding 'ideal cases'. We conclude that the on-demand protocol tolerates well a certain increase on contention between sources. For the remaining indicators, the on-demand has also a performance closer to that of the ideal first scenario. The on-demand protocol outperforms the pinwheel in this experiment, and the same trends observed before remain valid.

5.4. Comparison with existing protocols

The on-demand protocol has been designed for an important class of applications that result in non-uniform, bursty update arrival patterns. It takes advantage of the locality (both spatial and temporal) inherent to the bursty update arrival pattern to get superior performance. While such applications can be naturally supported by a token-based protocol, the on-demand token transfers together with

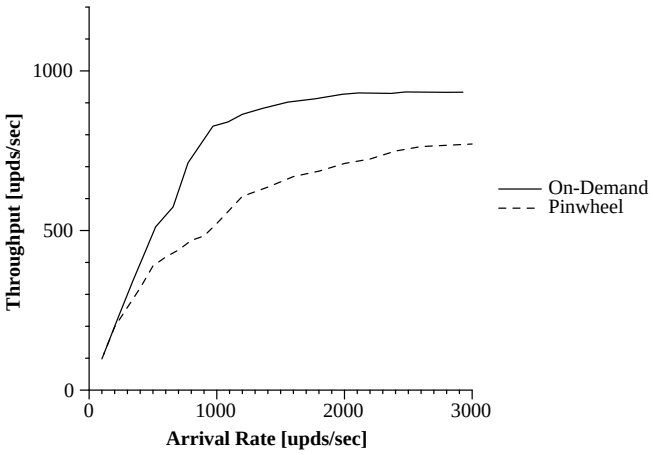

Figure 9. Throughput—partial burst overlapping.

Table 4. Delivery, stability, and messages—partial burst overlapping.

	On-demand			Pinwheel		
	$\lambda = 200$	$\lambda = 500$	$\lambda = 1000$	$\lambda = 200$	$\lambda = 500$	$\lambda = 1000$
Delivery (ms)	2.17	2.81	3.43	3.12	3.48	5.25
Stability (ms)	5.51	7.98	12.01	5.59	12.10	14.46
#messages	1.39	1.12	1.04	1.83	1.44	1.23

policies that ensure fairness and burst preservation are an efficient way of overcoming the deficiencies of moving-sequencer-based protocols that transfer the token following a fixed cyclical order. An example of a protocol with a fixed token circulation order is the Totem single-ring protocol [2]. Totem relies exclusively on the circulation of the token to disseminate the set of correctly received updates at each site; as a consequence, the node that originates a burst must wait until the token rotates twice around the ring in order to determine which updates have been received by the non-holders. The performance penalty grows with the size of the group, and this accounts for the significant difference between the on-demand's delivery/stability times and the numbers reported in [2]. While interesting for uniform update arrival rates, the flow control strategy used by Totem does not take advantage of the temporal locality of bursts, and requires that each group member have buffer space equal to twice the number of updates that can be broadcast *by every node* during a whole token rotation. In contrast, the on-demand protocol uses batched, non-blocking positive acknowledgments on a window size equal to the length of two sub-bursts from the same holder; this results in substantially smaller buffer requirements and better delivery and stability times, exploiting the burstiness of the arrival pattern.

The Reqtoken protocol [22] was independently published a few months after the initial publication of the on-demand protocol. Reqtoken and on-demand are based on a similar principle. Both of them benefit greatly from message batching. Unlike on-demand, Reqtoken does not guarantee a minimum token-holding time to the holder (thus increasing the probability of interrupting bursts before they finish), and it does not handle communication failures and stability information, depending on lower protocol layers for these tasks. Two interesting ideas used in Reqtoken are: having the holder place itself in the queue of token requestors before relinquishing the token (as a heuristic for a future potential burst), and using a timeout-based batching policy.

6. Batching of updates

In the protocol families described in the previous sections, as well as in the experiments and simulations reported, each update broadcast from a group member resulted in the transmission of a separate message in the underlying communication network. However, there are several reasons to believe that *batching* more than one update per network message can result in a substantial performance gain:

- Since the control and header information of the protocols uses a fixed amount of storage, fewer and larger messages will amortize this overhead over a larger payload and result in fewer bytes effectively being sent. For example, batching 10 updates in each message results in an update dissemination message of 55 bytes, 40 of which are user information—a payload/size ratio of 72.73%, previously equal to 23.53% for the non-batching version. Furthermore, these figures only take into account the control information of the on-demand implementation; the improvement is still larger when the lower-level UDP, IP, and 802.3 headers and trailers are considered. This results in a smaller per-update overhead, both in the physical transmission through the network and in the buffer copies

between different levels of the protocol stack at the sender and receiver nodes.

- A single system call to the underlying communication service is amortized over many updates. This argument, together with the previous one, results in a lighter load on the sending and receiving CPUs.
- Network congestion is smaller because fewer messages are being sent. This effect is especially important in common-medium networks such as Ethernet, where many attempts to gain access to the medium cause a significant performance degradation.
- For some protocols, less ordering information needs to be sent because updates contained in the same message are implicitly ordered. For example, it is sufficient to assign a single ordinal to the entire batch in the pinwheel and on-demand protocols.

We proceed to discuss alternative ways of batching many updates per message, and to provide sample performance figures that support the achievable improvement. Each broadcast server groups several updates originated locally in batches and packs each batch into a single message that is treated as a unit by the protocol. As a first policy, the broadcast server process can consume all the updates present in its input queue upon activation and then become inactive again. If the update that caused the activation of the broadcast server was sent by a local client at time t and the server process became active at time $t + \sigma$, the batch will contain the initial update plus all the ones arrived in the interval $(t, t + \sigma)$—where σ is the operating system's scheduling delay. However, σ is a random time interval, and is unbounded for non-real-time schedulers; even if σ was bounded, it might be too small to accumulate a good number of updates. An alternative deterministic policy consists of waiting a fixed number ω of time units between the arrival of the first update and the construction of the batch, as reported in [22]. If a client inserts an update in an empty buffer at time t, it schedules a timeout event that will awaken the broadcast server at time $t + \omega$. When the server wakes up, it builds a batch with all the updates present in the input buffer. A third alternative consists of building batches with a fixed number *batch_size* of updates (plus a timeout to send out the batch at low arrival rates), independently of the contents of the input queue when the server process is activated.

We illustrate the effect of the third policy on the performance of the on-demand protocol, batching together *batch_size* $= 10$ user updates in each update dissemination message. Figure 10 shows the throughput of the on-demand protocol's batching version for the three traffic patterns previously examined.

The throughput has improved dramatically by batching the updates. The best throughput attainable is achieved, as expected, for the scenario in which bursts are disjoint in time: 5680 upds s^{-1} for an arrival rate of 8064 upds s^{-1}. This is a fivefold improvement over the non-batching version of the protocol. Furthermore, the maximum update arrival rate that can be supported without degrading the performance is four times larger than in the non-batching case. Even in the worst case (full burst overlapping), the throughput is more than three times better than the non-batching case. The limiting factors are the overhead of buffer copying between protocol levels in the source and destination machines, and a larger fraction of lost messages (larger messages are less likely to fit in system buffers).

The average delivery times, stability times, and the number of messages exchanged per update broadcast are shown in table 5 for a sample arrival rate of 1000 upds s^{-1}. Both the delivery times and the stability times have improved marginally over the non-batching version. There are two conflicting effects that impact the delivery and stability times: the critical path of any given update is shorter because fewer system calls per update are performed, but there is a longer wait for the batch to complete before sending it to the network.

As expected, the number of messages exchanged per broadcast has decreased dramatically. The number of messages exchanged per broadcast is seven times less than that in the non-batching version in the best case (disjoint bursts), and five times less in the worst case (full burst overlapping). Thus the batching of updates improves all performance indexes that we consider important.

7. Summary

The performance of an atomic broadcast protocol under diverse operating conditions is crucial for the efficiency of the applications that use the atomic broadcast service. We have described three asynchronous atomic broadcast protocols that provide good overall performance at low as well as high update arrival rates, and in the presence as well as absence of communication failures. These protocols can tolerate omission and performance failures of the communication subsystem, and crash and performance failures of the processors. The protocols provide total order delivery and majority agreement semantics.

The elaboration of the pinwheel protocols was done as a result of studying the performance and combining the relative strengths of fixed-sequencer- and token-based existing protocols. The Pinwheel protocols substantially simplify and improve the idea of a rotating sequencer. They introduce novel flow control techniques that improve the performance at high arrival rates, as well as empty decision and newsmonger messages that improve the stability and message loss detection times when updates arrive slowly. Our simulation results show that the pinwheel protocols have broadcast delivery times in the absence of communication failures comparable with the Amoeba and Isis protocols, and that they are consistently better than the CM protocol. The number of messages sent to broadcast an update is also the same in broadcast-channel-based networks. Where the pinwheel protocols win is in providing better stability and message loss detection times, distributing the load evenly among group members, and improved delivery and stability times when updates arrive very fast. The good performance of the pinwheel protocols observed in the simulation is confirmed by performance measurements on an implementation of these protocols on top of the Unix operating system.

The on-demand protocol is the first published protocol designed to support bursty update arrival patterns. Previously existing atomic broadcast protocols do not take

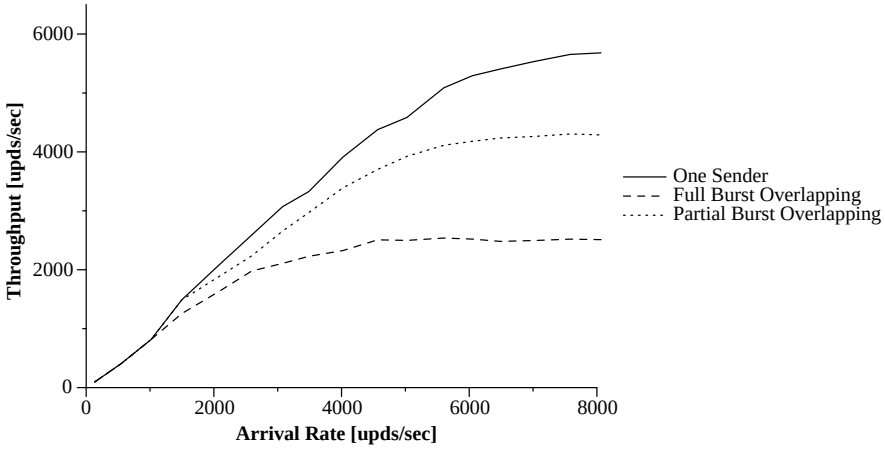

Figure 10. Throughput—update batching version.

Table 5. Delivery, stability, and messages—update batching version.

	Disjoint bursts	Full overlapping	Partial overlapping
Delivery (ms)	2.40	8.12	3.09
Stability (ms)	7.67	13.45	8.32
#messages	0.15	0.21	0.18

advantage of the special characteristics of non-uniform arrivals. The on-demand protocol takes advantage of the spatial and temporal localities of bursty arrivals, and of the fact that only a few of the group members need to initiate broadcasts, to get good performance independently of group size. We implemented the on-demand protocol on a network of workstations, and confirmed its performance advantage over traditional alternatives optimized for uniform arrivals. According to our measurements, the on-demand is the protocol of choice even when update arrivals are moderately bursty (i.e. there is some overlapping between bursts originated at different processors).

We also discussed how batching many updates per network message affects the on-demand protocol's performance. By definition, this technique is very well suited for bursty update arrival patterns and indeed results in a dramatic overall performance improvement for the on-demand protocol; all performance indexes are improved. Batching is useful for the pinwheel protocols also when updates arrive at a very fast rate, as it reduces the network congestion.

Acknowledgments

This work was partially supported by the Air Force Office of Scientific Research, Sun Microsystems, the Powell Foundation, the Microelectronics Innovation and Computer Research Opportunities in California, the Organization of American States, and the San Diego Supercomputing Center.

References

[1] Alvarez G, Cristian F and Mishra S 1995 On-demand asynchronous atomic broadcast *5th IFIP Working Conference on Dependable Computing for Critical Applications (Urbana, IL)*

[2] Amir Y, Moser L E, Melliar-Smith P M, Agarwal D A and Ciarfella P 1995 The Toten single-ring ordering and membership protocol *ACM Trans. Comput. Syst.* **13** 311–42

[3] Birman K, Schiper A and Stephenson P 1991 Lightweight causal and atomic group multicast *ACM Trans. Comput. Syst.* **9** 272–314

[4] Carr R 1985 The Tandem global update protocol *Tandem Syst. Rev.*

[5] Chandra T, Hadzilacos V, Toueg S and Charron-Bost B 1996 On the impossibility of group membership *Proc. 15th ACM Symp. on Principles of Distributed Computing* pp 322–30

[6] Chang J and Maxemchuk N 1984 Reliable broadcast protocols *ACM Trans. Comput. Syst.* **2** 251–73

[7] Cristian F 1989 Probabilistic clock synchronization *Distrib. Comput.* **3** 146–58
An earlier version, Cristian F 1988 *IBM Research Report (San Jose, RJ 6432)*

[8] Cristian F 1991 Asynchronous atomic broadcast *IBM Tech. Disclosure Bull.* **33** 115–16 presented at 1990 *1st IEEE Workshop on Management of Replicated Data (Houston, TX)*

[9] Cristian F 1991 Understanding fault-tolerant distributed systems *Commun. ACM* **34** 56–78

[10] Cristian F 1996 Group, majority, and strict agreement in timed asynchronous distributed systems *Proc. 26th Int. Symp. on Fault-tolerant Computing (Sendai)* pp 178–87 also available via anonymous ftp at cs.ucsd.edu as /pub/team/groupMajorityAndStrictAgreement.ps.Z

[11] Cristian F 1996 Synchronous and asynchronous group communication *Commun. ACM* **39** 88–97

[12] Cristian F, de Beijer R and Mishra S 1994 A performance

comparison of asynchronous atomic broadcast protocols *Distrib. Syst. Engng* **1** 177–201
[13] Cristian F and Fetzer C 1995 Fault-tolerant external clock synchronization *Proc. 15th Int. Conf. on Distributed Systems (Vancouver)*
[14] Cristian F and Fetzer C 1997 The timed asynchronous system model *Technical Report* CS97-519, UCSD available via anonymous ftp at cs.ucsd.edu as /pub/team/timedAsynchSystemModel.ps.Z
[15] Cristian F and Mishra S 1994 Automatic service availability management in asynchronous distributed systems *Proc. 2nd Int. Workshop on Configurable Distributed Systems (Pittsburgh, PA)*
[16] Cristian F and Mishra S 1995 The Pinwheel asynchronous atomic broadcast protocols *Proc. 2nd Int. Symp. on Autonomous Decentralized Systems (Phoenix, AZ)*
[17] Cristian F, Mishra S and Alvarez G 1995 High performance asynchronous atomic broadcast *Technical Report* CSE95-450, UCSD available via anonymous ftp at cs.ucsd.edu as /pub/team/highPerformanceAAB.ps.Z
[18] Cristian F and Schmuck F 1995 Agreeing on processor-group membership in asynchronous distributed systems *Technical Report* CSE95-428, UCSD
[19] Fetzer C and Cristian F 1995 On the possibility of consensus in asynchronous systems *Proc. 1995 Pacific Rim Int. Symp. on Fault-tolerant Systems (Newport Beach, CA)*
[20] Fetzer C and Cristian F 1997 A highly available local leader service *Proc. 6th IFIP Int. Working Conf. on Dependable Computing for Critical Applications (Grainau)*
[21] Fischer M J, Lynch N A and Paterson M S 1985 Impossibility of distributed consensus with one faulty process *J. ACM* **32** 374–82
[22] Friedman R and van Renesse R 1995 Packing messages as a tool for boosting the performance of total ordering protocols *Technical Report* TR95-1527, Cornell University Computer Science
[23] Kaashoek M F, Tanenbaum A, Hummel S F and Bal H 1989 An efficient reliable broadcast protocol *Operating Syst. Rev.* **23** 5–19
[24] Melliar-Smith P M, Moser L E and Agrawala V 1990 Broadcast protocols for distributed systems *IEEE Trans. Parallel Distrib. Syst.* **1** 17–25
[25] Peterson L L, Buchholz N C and Schlichting R D 1989 Preserving and using context information in interprocess communication *ACM Trans. Comput. Syst.* **7** 217–46
[26] Ramanathan S, Rangan V and Vin H 1992 Optical communication architectures for multimedia conferencing *Proc. 12th Int. Conf. on Distributed Computing Systems (Yokohama)* pp 46–53
[27] Sabel L and Marzullo K Election vs consensus in asynchronous systems *Technical Report* TR95-1488, Cornell University