The Timewheel Asynchronous Group Communication Protocol

Christof Fetzer, Shivakant Mishra and Flaviu Cristian
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093–0114, USA *

Abstract

We propose a membership and a reliable broadcast protocol that support several different semantics simultaneously and provide good overall performance. The broadcast protocol supports three atomicity semantics—*weak*, *strong*, and *strict atomicity*—and three ordering semantics—*unordered*, *total order*, and *time order*. These result in nine group communication semantics and an application can dynamically choose the desired atomicity and order semantics when broadcasting an update. These protocols provide fast delivery and stability times, use a small number of messages to accomplish a broadcast or a membership change, distribute evenly the processing load among group members, and use efficient flow control techniques. The good overall performance is maintained in the absence of any failures and normal update arrival rates, in the presence of communication or process failures, and when updates arrive at a very fast or very slow rate.

1 Introduction

With the ever increasing use of computers in everyday life, particularly in critical applications, the need for computer system dependability is increasing. Unfortunately, building dependable systems is complicated. Perhaps the main reasons for this complexity are the asynchronous nature of modern computing systems and the difficulties of handling failure occurrences. One technique to ensure dependability of a service is to implement it by a *team* of replicated servers. The underlying idea is that the currently running team members, that is, the current *group* of servers that implement the service, maintain a *consistent* replicated service state and, if one group member fails, the others form a new group and continue to provide the service. Group communication services are a set of fault-tolerant services that enable replicated application processes to maintain a consistent replicated state despite random communication delays and failures.

Reliable broadcast and *group membership* services are group communication services that have been shown to be useful for maintaining a consistent replicated service state [10]. A reliable broadcast service enables group members to disseminate team state *updates* so that, updates are delivered at all group members. The *membership* of the group that implements the service at a given point in time shrinks with failures and grows with process restarts. A group membership service provides a consistent system-wide view of which team members are operational at any given moment in time.

*Technical Report CSE95-411, 1995. This work was supported by the Air Force Office of Scientific Research, Sun Microsystems, the Powell Foundation and the German Academic Exchange Service (DAAD). S. Mishra is now with the Department of Computer Science, University of Wyoming, Laramie, WY 82071-3682

The main research issues in designing these protocols involve the exact semantics that should be provided and how to accomplish a good overall performance for the supported semantics. On one hand, not all applications need strong semantics, e.g. total ordering of updates, and in this case application designers would like to trade strong semantics for a better performance [6]. On the other hand, when applications need stronger semantics it is difficult and expensive to implement these on top of a reliable broadcast system with weak semantics; it is less expensive—with respect to the performance and cost of implementing it—to provide weak and strong semantics simultaneously by one protocol.

In this paper, we present a reliable broadcast and group membership protocol: the Timewheel group communication protocol. The Timewheel group communication protocol supports multiple semantics simultaneously: three different kinds of ordering of updates—*unordered*, *total ordered* and *time ordered*—and three different kinds of atomicity—*weak*, *strong* and *strict atomicity* [14]. This protocol provides good performance for all semantics at high as well as low update arrival rates, in the absence as well as presence of communication or process failures. Time ordered updates, i.e. the delivery of updates is ordered according to the send timestamps, together with strict atomicity, i.e. a process delivers an update only when a majority of the processes do this, allows to guarantee causal order of updates in the presence of "hidden" communication channels [17]. To provide these different semantics, particularly to preserve the order arising due to hidden communication channels and to detect process failures, we make use of an internal clock synchronization service that synchronizes local clocks of different nodes in the system [7, 12]. To ensure good overall performance, we allow the same messages to be used by the internal clock synchronization, group membership, and reliable broadcast protocols.

The Timewheel protocol is unique in several respects. First, it provides a wide range of semantics allowing the clients to choose the right semantics for their needs. Second, to the best of our knowledge, this is a first attempt in preserving causal order arising due to "hidden" channels in an asynchronous fault-tolerant system. Third, to the best of our knowledge, this is a first attempt in using a clock synchronization service to design group communication protocols in an asynchronous distributed system. In addition to preserving the causal order arising due to "hidden" communication channels, the use of a clock synchronization service also simplifies the design of the Timewheel protocol. For example, the Timewheel group membership service uses clock synchronization to detect process failures and to handle failure events. Finally, the Timewheel group communication protocol provides good overall performance in the absence as well as presence of communication or process failures. The Timewheel broadcast protocol provides fast delivery and stability times, use a small number of messages to accomplish a broadcast, distribute evenly the processing load of ordering messages among group members, and use efficient flow control techniques to provide good performance at high as well as low update arrival rates. The Timewheel membership protocol does not exchange any messages during normal operation (when there are no failures), minimizes the possibility of falsely detecting a failure, minimizes interference with the broadcast protocol, and uses a small number of messages to form a new group when some processes fail or recover.

The rest of the paper is organized as follows. We start by defining our system model and assumptions. We then give an overview of the semantics provided by the Timewheel protocol and the the key ideas underlying the Timewheel protocol. Finally, we present the broadcast and membership protocols in detail.

2 System Model and Failure Hypotheses

We assume a distributed system in which multiple nodes are connected by a communication network. Nodes communicate by exchanging messages through the communication network. We do not make any assumption about the communication network topology: it can be either a broadcast channel or a point-to-point network.

The protocols to be presented depend on an *asynchronous* datagram communication service, such as UDP, which does not guarantee any bound on message transmission delays. However, we assume that a a timeout delay TD is associated with each message. We say that a process has received a message in a *timely manner* when the transmission delay of the message was not greater than TD. A message may be lost or take more time to arrive than the timeout delay associated with it, but we assume that any message received is received uncorrupted. In other terms, we assume a datagram service with *omission/performance* failure semantics [9]. The overwhelming majority of local or wide area network communication systems provide datagram services with the above failure semantics, and so our protocols can be implemented on top of a large variety of existing low-level communication services.

We also assume that the local clocks of different nodes are *synchronized*, i.e. the *maximum deviation* between two correct clocks is bounded by a given constant δ. One approach to bound the deviation is to synchronize all clocks to an external time signal [7]. Another approach is to synchronize the clocks by an internal clock synchronization algorithm [12]. In our system model, an internal clock synchronization algorithm cannot always bound the deviation between correct clocks by an a priori given constant. However, the clock synchronization service proposed in [12] allows a node to determine when its clock may not be synchronized within the given maximum deviation with the clocks of other correct nodes. We assume that a node behaves as if it has crashed whenever it detects that its clock may not be synchronized.

A *team* is a set of processes. A group membership protocol provides a view of which processes in a team are currently operational. For simplicity, we consider in the following only one team and we use the term *process* to refer to a process in the given team. Every process p has a unique rank which we denote by $rank(p)$. We assume that at any time, there exist a majority of processes S so that all processes in S where members of the last group created and they survive until a new group is created.

3 Timewheel Group Communication Protocol

The Timewheel group communication protocol consists of a group membership and a reliable broadcast protocol for an asynchronous distributed system. The motivations for designing these two protocols are to have good overall performance and to support different protocol semantics. The Timewheel protocol supports different semantics simultaneously. This allows users to optimize the performance of the communication subsystem by using weaker semantics, whenever feasible.

3.1 Semantics

The protocol semantics that we consider belong to two classes: *atomicity* and *order*. The atomicity property specifies if an update or a membership change is delivered, while the *order* property determines the order in which different updates and membership changes are delivered. The Timewheel group communication protocols provide three atomicity semantics for the delivery of updates—*weak atomicity*, *strong atomicity* and *strict atomicity*—and three order semantics—*unordered*, *total order*, and *time order*. The delivery of membership changes are always time ordered, but a process can choose between strong or strict atomicity. Each group contains at least a majority of the processes.

Group communication with causally ordered delivery has been described before as a facility to construct fault-tolerant applications [2, 20, 3, 19]. Causally ordered delivery is defined by using the "happened before" relation [17]. An event a *happened before* another event c, iff

- a and c are events in the same process, and a occurs before c,
- a denotes the event when a process communicates some information I and c denotes the event when a process gets information I, or
- there exists an event b, such that a happened before b and b happened before c.

Causal delivery ensures that if proposal of an update u_1 happened before the proposal of an update u_2 and u_2 is delivered, then u_1 is delivered before u_2.

However, processes are not necessarily restricted to communicate only by exchanging messages using the group communication service; they could use other mechanisms such as a file system to communicate. These other communication facilities are called *hidden channels*. Causal delivery of updates requires that when p proposes u_1 and then communicates information I to process q using a hidden channel, and after q gets I, q proposes u_2, then u_2 cannot be delivered before u_1. Synchronized clocks can be used to guarantee causal delivery in the presence of hidden channels; a requirement for this is that the minimum communication delays of hidden channels is greater than the maximum deviation between clocks (see [17] for details).

Total order delivery ensures that all group members deliver all updates with total order requirement in the same order and that this order preserves the FIFO property, i.e. when a process proposes update u_1 and then u_2, then u_2 cannot be delivered unless u_1 has been delivered. Time order delivery ensures that updates are delivered in the order of their send timestamps, where the send timestamp of an update is the local clock time when the broadcast of that update is initiated. Time ordered delivery guarantees total order between updates delivered to different groups. Consider the case that processes p and q are both in teams T_1 and T_2, and update u_1 is delivered to T_1 while update u_2 is delivered to T_2: when u_1 and u_2 are sent with total order requirement, then p could deliver u_1 before u_2 while q could deliver u_2 before u_1; when u_1 and u_2 are sent with time order requirement, then p and q have to deliver u_1 and u_2 in the same order. Finally, unordered delivery allows group members to deliver updates in any order and potentially, different group members may deliver updates in different orders.

Weak atomicity requires that when a group member p delivers an update u then either eventually all group members will deliver u, or eventually a new group g is created that does not contain p as a member and no member of g has delivered u. Weak atomicity allows the following behavior. A group member p proposes an update u_1 and then crashes. Because of communication failures, only group member q receives update u_1. After delivering u_1, q proposes an update u_2 and then crashes. All group members deliver u_2, but not u_1; notice that this can happen for all ordering requirements. This may lead to an inconsistent system state, where group members have delivered an update u_2 without delivering an update u_1 on which u_2 potentially depends. In particular, weak atomicity does not guarantee causal delivery.

Strong atomicity requires that when a process p delivers an update u then either eventually all group members deliver u, or eventually a new group g is created that does not contain p as a member and no member of g has delivered u, and all updates proposed by p after delivering u are not delivered by any member of g. To illustrate this, let us assume that no recoveries occur and all updates are delivered in total order: there exists a sequence H of updates and membership changes such that a majority of processes deliver these updates and membership changes in the order given by H; only updates proposed by processes which have delivered updates and membership changes according to H can be in H.

Strong atomicity prohibits the inconsistent behavior described for weak atomicity. However, strong atomicity allows a similar behavior if processes can use hidden channels to communicate. For example, suppose a process p proposes an update u_1 and then crashes. Because of communication failures, only process q receives update u_1. Process q delivers u_1, uses a hidden channel to transmit information I to process r and then crashes. Process r proposes an update u_2 which causally depends on I (and hence on u_1), but all group members deliver u_2 without deliverying u_1. This means that strong atomicity cannot guarantee causal delivery in the presence of hidden channels.

Strict atomicity requires strong atomicity and that a process p delivers an update u only when a majority of processes do this. The inconsistent behaviors described for weak atomicity and strong atomicity, where processes may deliver an update u_2 that causally depends on update u_1 without delivery u_1, cannot happen in strict atomicity. This is because a process q can deliver u_1 only if a majority of processes deliver $u1$ and hence all group members deliver $u1$. Otherwise no process can deliver u_1 and in particular no process can propose an update u_2 which causally depends on u_1. Since the Timewheel group communication service assumes synchronized clocks, time ordered delivery together with strict atomicity guarantees causal delivery in the presence of communication through hidden channels as well.

The following properties are provided by the Timewheel group communication protocol for all atomicity and order semantics (see [14] for a formal description). Only updates proposed by group members are delivered and if a process p omits to deliver an update, which is delivered by all other group members, then p is removed from the membership. If a communication partition occurs, only the partition containing a majority of processes is allowed to make progress, i.e. no updates and membership changes are delivered by processes belonging to a minority partition.

3.2 Performance Metrics

We consider the following important performance indexes: average broadcast delivery time, average broadcast stability time, average number of messages needed to broadcast an update, and the distribution of processing load among group members. The broadcast *delivery time* is defined to be 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 messages can get lost, the broadcast servers that implement the distributed broadcast service must store a message that is being broadcasted in a local buffer until they learn that the message is *stable*, that is, it was received by all group members. The broadcast stability time is the duration between the moment a broadcast server receives an update to be broadcasted and the moment all broadcast servers learn that the update is stable. Stability is important, because it is a measure of how long a message stays in the buffers of group members and how much buffer space a protocol needs. The *number of messages* needed to broadcast an update includes all messages sent by different group members to complete the broadcast of the update. 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).

3.3 Protocols

The three atomicity semantics and the three order semantics result in nine Timewheel group communication semantics. A novel feature of the Timewheel protocols is that applications can dynamically choose atomicity and order semantics when broadcasting an update. As a result, a client can choose to broadcast one update with one pair of atomicity and order semantics and another update with another pair of atomicity and order semantics. The exact ordering semantics when different order semantics are intermixed in an application are described in Section 4.2.

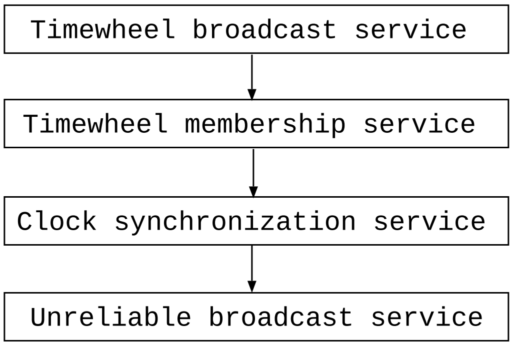

Figure 1: System Architecture of Timewheel Group Communication Protocol.

The system architecture of the Timewheel group communication services is shown in Figure 1. The Timewheel broadcast protocol is an extension and optimization of the Pinwheel atomic broadcast protocols developed at the University of California, San Diego [13]. As in [13], the Timewheel broadcast protocol uses a single process, called *decider*, to order the delivery and to establish the stability of broadcast updates; as observed in [11], such a technique provides good delivery and stability times. In order to distribute the processing load evenly among all group members and to detect process or communication failures fast, the role of the decider is rotated among group members in such a way that there is at most one decider in the group at any point in time. In contrast to other rotating decider protocols like [5, 13] the Timewheel protocol uses positive acknowledgements instead of negative acknowledgments to simplify and to speed up the exclusion/inclusion of processes from/to the membership and the establishment of message stability. By batching multiple acknowledgements into one message the Timewheel protocol can achieve a messages per update ratio at least as good as the negative acknowledgment schemes.

The membership protocol use a similar approach to order membership changes and provide the required atomicity: membership changes are treated similar to updates, except that only the decider can propose membership changes. A crash of a process results in the loss of the decider role and is masked by an election of a new decider which removes the crashed process(es) from the membership and creating a new group. The membership protocol is optimized for the common case: it uses separate election protocols to mask single failures and multiple failures. When multiple failures occur, the global time base established by the clock synchronization service is used in the election of a new decider.

Both the group membership and clock synchronization protocols need to send messages periodically to detect process crashes and to synchronize clocks respectively. The broadcast protocol also needs to send messages regularly to order the delivery and establish the stability of broadcast updates, and to rotate the role of decider. The Timewheel group communication service minimizes the number of messages sent by using the same message to convey update ordering/stability information, detect processor crashes, and synchronize clocks. In addition, each protocol uses several mechanisms to further reduce the number of messages exchanged, and at the same time, maintain good overall performance in the presence as well as absence of failures.

4 Timewheel Broadcast Protocol

In this section we describe the Timewheel broadcast protocol. For simplicity, we will not describe all optimizations that we use to reduce the computation time. The Timewheel broadcast protocol is an optimization and extension of the protocols proposed in [5] and [13]. The Timewheel protocol retains good performance of the Pinwheel protocol [13] in the absence of communication failures, improves performance in the presence of communication failures, supports different delivery semantics, and provides functionality to deal with process failures without interfering with the broadcast of updates.

4.1 Protocol Description

To propose an update u at local time ts with order ord and atomicity ato, a member s in group g broadcasts a *proposal* message $< g, u, s, n, ts, hdo, hcro, ord, ato >$ to all other group members; ts is called the send timestamp of u and s is the proposer of u. Update u is uniquely identified by the rank s of the proposer and a monotonically increasing sequence number n (initially $n = 0$). Proposal dissemination is done via either a multicast message in a broadcast network, or $N - 1$ point-to-point messages in a point-to-point network.

A unique group member called *decider* orders updates/membership changes by assigning a unique natural number, called *ordinal*, to each update/membership change; this ordering is independent of the atomicity and order semantics required by the client. The first update/membership change ordered is assigned ordinal 0, the next is assigned ordinal 1, and so on. The highest delivered ordinal, hdo in a proposal message, specifies that the proposer has delivered no updates with ordinals greater than hdo; this field is used to satisfy strong/strict atomicity. The highest continuously received ordinal, $hcro$ in a proposal message, specifies that the proposer has received all proposals/membership changes with ordinals smaller than or equal to $hcro$; it is used to ensure smaller stability times.

The Timewheel broadcast protocol uses a positive acknowledgement scheme that batches acknowledgements to reduce the number of acknowledgement messages. A decider orders and explicitly acknowledges proposals by sending a *decision message*. Other processes implicitly acknowledge the receipt of a proposal with ordinal o when they send a proposal with $hcro$ greater than or equal to o. To bound the stability time, a process must acknowledge all received proposals periodically. To facilitate this and to distribute evenly the communication overhead of sending decision messages among group members, the role of decider is rotated between group members. To do this, a logical ring that defines a predecessor/successor relation among group members is formed. A group member becomes a decider after it receives a decision message from its predecessor in the logical ring, and relinquishes its decider role by sending a decision message. The duration for which a process can assume the role of the decider before it has to relinquishes this role is bounded by an *a priori* given constant D. For example, if group member p is the current decider and member q is its successor and p relinquishes its decider role at ts by sending a decision message, then q assumes the role of decider when it receives this decision message from p. Furthermore, q has to send its decision message before time $ts + D$. Since a group member can play the role of the decider only after receiving a decision message from its predecessor, there can be *at most one* decider at any point in time.

A decision message is a tuple $< ord, oal >$, where the *ordering and acknowledgement list*, oal, is a list of descriptors of updates and membership changes used to acknowledge and order proposals/membership changes. The ordinal, ord, specifies the ordinal of the first entry in oal. A *proposal descriptor* $< s, n, ts, ack >$[1] identifies an update with id $< s, n >$ sent at time ts by group member s. The acknowledgement vector, ack, is a bit vector containing one bit for each group member; $ack[i] = 1$ implies that member i has received the update proposed in proposal with id $< s, n >$. A *membership descriptors* $< g, ack >$ identifies a group g created by the membership protocol. Once again, ack is a bit vector containing one bit for each group member; $ack[i] = 1$ implies that member i has received this group.

[1] Actually, it would be sufficient to include only s and ack in a proposal descriptor. For simplicity, we include ts and n in proposal descriptors in our discussion.

In the following we use oal to refer to the ordering and acknowledge list of the current decider or the oal included in the last decision message when there is no current decider. We use $field\ oal$ to refer to the field oal in a specific decision message. A decider can modify the oal by deleting stable proposals/membership changes from the head of this list and appending new descriptor at the end of this list. The ordering of proposal descriptors in oal and ord implicitly associate an ordinal with each descriptor in oal; the ordinal associated with the kth descriptor in oal is $ord + k$; position numbers in oal start at 0. Hence, ord also specifies the number of elements removed from oal so far.

Each member maintains a *proposal buffer*, *pb*, to store the received proposals and a *proposal descriptor buffer*, *pdb*, to store proposal descriptors and their ordinals. When a process p receives a proposal $< g, u, s, n, ts, hdo, hcro, ord, ato >$, p stores this proposal in its *pb* and, if a proposal descriptor for this proposal doesn't exist in its *pdb*, it stores a proposal descriptor $< s, n, ts, ack >$ with an *undefined* ordinal; $ack[p]$ and $ack[s]$ are set to one and all other ack bits are set to zero. In case a proposal descriptor for this proposal already exists in *pdb*, p sets $ack[p]$ to 1 and the timestamp to ts in this descriptor. Communication failures may create *gaps* in *pdb*, i.e. for some member s, proposal descriptors for proposal ids $< s, n >$ and $< s, n + m >$ $(m > 1)$ may be present in *pdb*, but not the descriptors of proposal ids between $< s, n+1 >$ and $< s, n+m-1 >$ sent by s. In such a case, p fills these gaps by storing proposal descriptors for all proposal ids between $< s, n >$ and $< s, n + m >$; in all each of these descriptors the send timestamp and the ordinal is undefined, $ack[s]$ is set to 1, and all other ack bits are set to zero. In addition, each group member s maintains an array $mhcro$ that records for each member r the maximum $hcro$ received so far from r.

On receiving a decision $< ord, oal >$ from the last decider d, a process p does the following. It checks if d has missed one of p's proposals, i.e. if there exists one or more proposal descriptors of p's proposals in field oal with $ack[d] = 0$. In case d has missed a proposal sent by p, p retransmits that proposal to d; in case d has missed a proposal pr from a sender s which is not a member of the current group and p is the first "successor" of s which has acknowledged pr, then p will retransmit pr. For each proposal pr whose descriptor is present in field oal and in p's *pdb*, p does the following: (1) it updates the ack vector of the proposal descriptor of pr in its *pdb* by performing a bitwise OR operation with the ack vector of the proposal descriptor of pr in field oal, (2) if the timestamp uts of this descriptor is undefined in its *pdb*, it sets uts to the timestamp of the corresponding descriptor in field oal, and (3) it calculates and stores the ordinal of pr in *pdb*. For each proposal pr whose descriptor is present in field oal but not in p's *pdb*, p stores that proposal descriptor in its *pdb* along with its ordinal, and fills any gaps, that might have been created, as described earlier. In addition, if p is the successor of d it assumes the role of the decider.

When a process d becomes the decider by receiving a decision $< ord, oal >$, it first uses field oal to update its *pdb* as described above. Before sending a decision message and hence transferring the role of decider, it updates oal as follows. It acknowledges all membership changes included in oal by setting $ack[d]$ bit of all membership descriptors to 1. It also acknowledges each proposal in oal that d has received. For each process r, it sets $ack[r]$ to 1 in all proposal descriptors in oal of proposals proposed by r whose ordinal is less than or equal to $mhcro[r]$.

Because of retransmissions, a process can receive a proposal messages more than once. To guarantee that an update is delivered at most once, a process has to keep track of duplicated proposal messages. Furthermore, an update proposed by a group member should be admitted to be delivered only if it has been a member of all groups formed since it proposed that update. For example, let process q be a member of groups g_0 and g_2, but not a member of group g_1, and the groups were created in the order g_0, g_1, and g_2. Then no update proposed by q before it joined g_2 is allowed to be delivered by a process r after r has delivered the membership of g_2. In particular, an update proposed by q when it was a member of g_0 is not allowed to be delivered by a member r after r has delivered the membership for group g_1. To ensure this property we introduce now two "deletion rules" which delete such proposals from the *pd* and *pdb*.

Each process r maintains a table $tsdm$ of lower bounds of send timestamps; $tsdm[q]$ is the maximum of the time when q last joined the group and the send timestamp of the last decision message sent by q. In the following, the *current group* refers to the last group described in *oal* or, if *oal* does not contain a group descriptor, the last group which was purged from *oal*. After the current decider d has set *ack* vectors of various proposal descriptors in *aol* as described earlier, it deletes each proposal descriptor *pd* sent by a process s from *pdb* and the corresponding proposal from *pb* according to the following *deletion rules*: (1) *pd* has an undefined ordinal and s is not in the current group, or (2) *pd* has an undefined ordinal and *pd*'s send timestamp is ≥ 0 but smaller than $tsdm[s]$.

After the current decider d has deleted all obsolete proposals it appends all remaining proposal descriptors which are in its *pdb* and which have an undefined ordinal number. In particular, it appends a proposal descriptor for each proposal it has sent, but which hasn't been included in *oal* so far. These proposals are appended in the order of their send timestamps and in the order of the ids of their proposers, if two proposals have the same send timestamps. d stores the ordinals of all proposal descriptors it appended in *pdb*, removes all descriptors from the head of *oal* which have been acknowledged by all current group members, and increments *ord* by the number of descriptors removed. Finally, d sends a decision message after either appending k or more descriptors in *oal*, or after D time units have elapsed since it assumed the decider role.

4.2 Delivery of Updates

A process p delivers an update u proposed by process s as soon as the *atomicity*, *order*, and *general delivery conditions* are satisfied. The general delivery condition has to ensure that process p does not deliver an update u which is later removed from the *pdb* by the above described deletion rules or which is later removed from the *oal* (see section 5.4). To ensure this, process p must not deliver an update u when (1) u is erasable by the deletion rules, or (2) u is marked as undeliverable (see section 5.4), i.e. the sender of u is suspected to have failed and p requested that u won't be delivered by the group.

The atomicity condition for an update u that was broadcasted with a highest delivered ordinal of *hdo* and a strong atomicity requirement is satisfied when each proposal *pr* whose associated ordinal is less than or equal to *hdo* has been received by p or p knows that a *majority* has received *pr*, i.e. there exit a majority of processes S and each process in S is a member of the current group and has delivered *pr*. In the absence of hidden channels, this condition ensures that process p has received all updates on which u could causally depend.

Each group member maintains an integer *hdo* (highest delivered ordinal), which is included in every proposal message sent. When a member p delivers an update u, it updates *hdo* by setting *hdo* to the maximum of *hdo*'s old value and the ordinal of u. For updates with unordered delivery semantics, it is possible that p does not know the ordinal of u when it delivers u. In this case, *hdo* is updated as soon as p knows the ordinal of u, i.e. when it receives a decision message containing the proposal descriptor of u. The semantics of the field *hdo* in a proposal message require that s has to delay the broadcast of a proposal *pr* with strong/strict atomicity, until it knows the ordinals of all proposals it has delivered so far, i.e. until *hdo* has been updated for all updates that p had delivered so far. This implies that *pr* gets a higher ordinal then all updates it may depend on.

The atomicity condition for an update u that was broadcasted with strict atomicity requires the validity of the condition for strong atomicity and that there exists a majority which has acknowledged the receipt of u. Group members determine if a majority has received update u by checking the *ack* vector of the proposal descriptor of u in *pdb*.

The atomicity condition for updates sent with weak atomicity is always true. Also the order condition for updates with unordered delivery semantics is always true. So, a group member can deliver an update u sent with weak atomicity and unordered delivery as soon as the general delivery condition for u is satisfied. The order condition for an update u with total order delivery is satisfied at p as soon as p has delivered all updates with unordered/total order delivery semantics that have ordinals smaller than the ordinal of u.

The order condition for an update u with time order delivery is satisfied when all updates with send timestamps smaller than that of u have been delivered by p. To do this, each member s maintains a timestamp vector $hsts$ (*highest send timestamp*). For any member r of the current group of any team T with s is a member of T, $hsts[r]$ is the highest timestamp ts, such that all proposals sent by r with send timestamps less than or equal to ts are known to s. Member s can know about a proposal sent by r by either receiving that proposal from r, or receiving a decision message that contains the proposal descriptor of that proposal. Since a group member stores proposal descriptors for each proposal received and for each proposal descriptor received in a decision message, it has a proposal descriptor in *pdb* for each proposal it knows about. The *order* condition for an update u with time order delivery and with send timestamp ts is true, iff (i) $(ts, s) \leq \min\{(hsts[r],r) : \forall r \}$, (ii) all updates with smaller send timestamps known to s have been delivered, and (iii) all unordered/total ordered updates with smaller ordinals have been delivered.

The ordering conditions imply that an unordered update u with ordinal o can be delivered before any updates with smaller ordinals. An update with total ordered delivery is delivered after all unordered/total ordered updates with smaller ordinals, but it can be delivered before time ordered updates with smaller ordinals. After a process p delivers a time ordered update u with send timestamp ts, p does not deliver any other update with a send timestamp smaller than ts.

4.3 Update Stability and Flow Control

Since proposal descriptors of all ordered but unstable proposals are included in a decision message, establishing a fast update stability is very important to reduce the size of a decision message. The Timewheel broadcast protocol employs several techniques to ensure a faster update stability. We already described the use of the decision messages and the inclusion of the *hcro* in a proposal message to reduce the stability time.

However, in a situation where one or more group members do not send any proposals and other group members send proposals at a fast rate, the decision message size may become large. To address such situations, the Timewheel broadcast protocol employs another technique. A group member p sends a *delivery update* message that includes its *hcro* to all group members, if (i) the number of proposal descriptors in the latest decision message received is greater than *mtr* (maximum threshold), (ii) the difference between the highest ordinal in the latest decision message and the *hcro* p reported last is greater than *mlg* (maximum lag), and (iii) if the maximum time until p will be the the decider is greater than L time units. This ensures that silent group members send their *hcro* if the decision message size is becoming large. This in turn ensures a faster update stability when some group members are sending proposals at a very slow rate while others are sending at a very fast rate.

In addition to ensuring small decision message sizes, update stability is also used in purging proposal buffers and proposal descriptor buffers. After an update u is delivered by r and becomes stable at r, r removes the proposal containing u from its *pb* and its proposal descriptor from *pdb*. Despite these rules for purging 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. The purpose of *flow control* is to handle such situations. The flow control therefore piggy-backs on every decision message the *highest continuously delivered ordinal*, *hcdr*, i.e. all proposal with ordinals not greater than *hcdr* have been delivered by the decider. An update u is said to be *delivery stable* at member p, when p knows that all members have delivered u. In the Timewheel broadcast protocol, the possibility of a buffer overflow is avoided by restricting group members from accepting new proposal requests from their local clients when the number of locally originated updates that are not yet known to be delivery stable is *modu* (*maximum own delivery unstable*). Therefore, a buffer overflow cannot occur at a group member if its proposal buffer can store at least N * *modu* proposals, and the proposal descriptor buffer can store N * *modu* proposal descriptor-ordinal pairs.

4.4 Qualitative Performance

The Timewheel broadcast protocol uses a positive acknowledgement techniques to detect proposal losses. The number of messages sent per update broadcast is very small, because this protocol batches multiple acknowledgements into one decision message and implicitly acknowledges multiple messages by sending a proposal message. Therefore, the number of messages sent per update is at least as good as that of rotating decider protocols with negative acknowledgement schemes [5, 11, 13].

The delivery time of a broadcast update depends on order and atomicity requirements. In case of total order and strong atomicity requirement, the delivery time 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 rotating-decider protocols [11, 13]. Hence, the delivery time of the Timewheel protocols with total order and strong atomicity requirements is comparable to that of the Pinwheel protocol which only provides total order/weak atomicity semantics. The delivery semantics for

Because the role of the decider rotates among group members even when the update arrival rate is low and group members use several techniques such as including acknowledgements in a decision message or sending delivery update messages, the stability time of updates is low. These techniques also reduce the delivery times in case of strict atomicity requirement. Furthermore, the continuous rotation of the role of the decider, irrespective of the update arrival rate, ensures a fast detection of communication failures when updates arrive at a slow rate. The flow control techniques described ensure that performance is good even when updates arrive at a very fast rate or when communication failures are frequent.

4.5 Multiplexing of Updates

The number of messages sent when updates arrive very fast can be substantially reduced by batching together several updates in a single proposal message. Update batching contributes to reduce the number of proposal messages as well as the size of decision messages, because now one single proposal descriptor orders and acknowledges multiple updates. We found batching very useful, since it also reduces the possibility of network congestion by reducing the number of messages transmitted.

5 Timewheel Membership Protocol

The goal of a membership service is to provide clients with an up-to-date view of what processes are operational. These views are provided to the clients by upcalls from the membership service. We use the expression *delivery of a membership change* to denote such an upcall. All correct processes must deliver membership changes in the same order and the relative ordering between the delivery of different updates and membership changes has to be the same, i.e. if a member delivers an update u before (after) delivering a membership change, all members should deliver u before (after) this membership change.

5.1 Overview

We conceptionally split the membership protocol into two parts—*failure detector* and *group creator*. Each failure detector maintains an *alive-list* of processes which are currently functioning correctly. The failure detector is unreliable, i.e. an alive-list can contain processes which have failed or there might exist some processes that are functioning correctly but not in the alive-list, and alive-lists of different failure detectors can be different. The *group creator* uses this alive-list to maintain a *group-list*. It guarantees that all correct processes in the group-list agree on the current and past group-lists. We call a process in the group-list a *group member*.

While the Timewheel membership protocol deals with all failure scenarios that may occur within the failure model assumed, it is optimized for those scenarios that are more likely to occur than others. This protocol does not sent any messages as long as all group members periodically send their decision messages. When the role of the decider is lost because of a single process crash or an omission/performance failure of a single decision message, a simple and fast *single failure* election protocol is started to instantiate a new decider. When the less likely case of multiple failures have occurred, a *multiple failure* election protocol is started to create a new group.

The membership protocol uses three *control messages*: *no-decision*, *join*, and *reconfiguration*. In addition, a decision message is also treated as a control message by the membership protocol. A failure detector keeps all processes under surveillance by checking that all processes periodically send control messages. It expects that the current decider d sends a decision message after at most D time units and that the successor of d sends a control message after at most D time units after d has sent its decision message, and so on. It suspects that a group member p has failed, if it doesn't receive a control message from p in the expected interval of time. The failure detector informs the group creator whenever it suspects that a process p has crashed. The group creator then uses a single-failure or a multiple-failure election protocol to instantiate a new decider and to exclude the failed process(es) from the membership.

The single failure election protocol works in the following way. If the successor p of the current decider d suspects that d has failed, p sends a *no-decision* message requesting that d should be removed from the membership. When a process r receives a no-decision message suspecting d from r's predecessor and if it agrees with this suspicion, it sends a no-decision message in at most D time units. The single failure election protocol terminates when all group members except d have agreed about the suspicion of failure of d. This happens when the predecessor q of d receives a no-decision message from its predecessor, and agrees with the suspicion. In this case, q removes d from the membership by appending a new membership descriptor in *oal* in which p is not a member. On the other hand, in case a process r receives a no-decision message suspecting the failure of d from its predecessor, and r has received the last decision message from d, i.e. r does not suspect the failure of d, r becomes the new decider, and sends decision message as usual.

The multiple failure election protocol uses a time-slotted approach to agree on the new membership. Each process p sends during its time-slots a *reconfiguration* message which contains p's current alive-list and the highest timestamp of a decision message p has sent or received. The process q proposing the highest timestamp can create a new group when at least a majority of processes have sent a reconfiguration message and were members of the last group q knows about.

The initial group, i.e. the first group created after the system starts, is formed in the following way. When a process is created or recovers after a crash, it sends a *join* message in each of its time-slots. It continues to update its alive-list based on which processes have sent join messages. When a majority of the processes have sent join messages with the same alive-list, a new decider is instantiated.

5.2 Failure Detector

As described earlier, the global time-base provided by the synchronized clocks is divided into cycles and the cycles are divided into slots; each process has exactly one slot per cycle. Let N denote the number of processes and FD_p denote the failure detector of process p. The alive-list of FD_p contains p and each process q that sent at least one *control message* in the last N slots, which p has received. A failure detector uses the send timestamps to maintain its alive-list.

The length of each time slot has to be at least as large as the maximum duration D a process can be the decider. During a failure free period all group members are therefore in the alive-list, i.e. no group member is suspected to have failed. Failure detector FD_p piggy-backs its alive-list represented by a bit vector on each control message process p sends. The group creator uses this information to create new group lists.

A failure detector also informs the group creator when it suspects that a process has failed. Consider the case when process p has received a decision message with send timestamp ts from the current decider d. After receiving this decision message, FD_p expects a control message from the successor e of d before time $ts + 2D$. If p does not receives a control message from e with a timestamp greater than ts before time $ts + 2D$, it informs the group creator that e is suspected to have failed.

5.3 Group Creator

It is the responsibility of the group creator to instantiate a new decider when the role of the decider is lost. The group creator ensures that all created groups include at least a majority of the processes and all members have the same group-list. This is done by allowing only the decider to change the group-lists. The decider disseminates these changes by appending a membership-descriptor containing the new group-list to the ordering and acknowledge list in its decision message. The group creator of other members uses this group-descriptor to update its group-list.

We describe the group creator as a finite state machine with six states (see Figure 2): *join, no-failure, 1-failure-receive, 1-failure-send, wrong-suspicion*, and *n-failure*. During failure-free periods all group members are in no-failure state, they all become decider periodically, and send at least one decision message per cycle. The 1-failure-receive/1-failure-send state is used to handle a single omission/performance failure of a decision messages (which may cause the loss of decider) or a single crash/performance failure of a current group member. The wrong-suspicion state is introduced to mask situations in which some processes do not receive a decision message from the current decider d in time and try to exclude d from the membership. When a process detects that more than one failure has occurred during one cycle, it changes to n-failure state to elect a new decider. The join state handles the creation of a new group that includes the members of the current group and processes which want to be integrated into the group.

The failure detector FD_p of each member p keeps all other processes under surveillance. When p receives a decision message from decider d with send timestamp ts, it expects to receive a control message from the successor s of d before time $ts + 2D$; in the following, we call s the *expected* sender. Also, we say that a *timeout* failure has occurred, when the failure detector notifies the group creator that a process is suspected to have failed. We assume that processes reject duplicate or old control messages, i.e. when we say that a process p receives a control message m from the expected sender s, we implicitly assume that p checks the send timestamp of m to determine that it hadn't already received m, and that m is sent after time ts.

When a process p is created or recovers from a crash, it is in join state. In this state, it sends in each of its time-slots a *join* message requesting that it wants to be a member of a new group. If a process

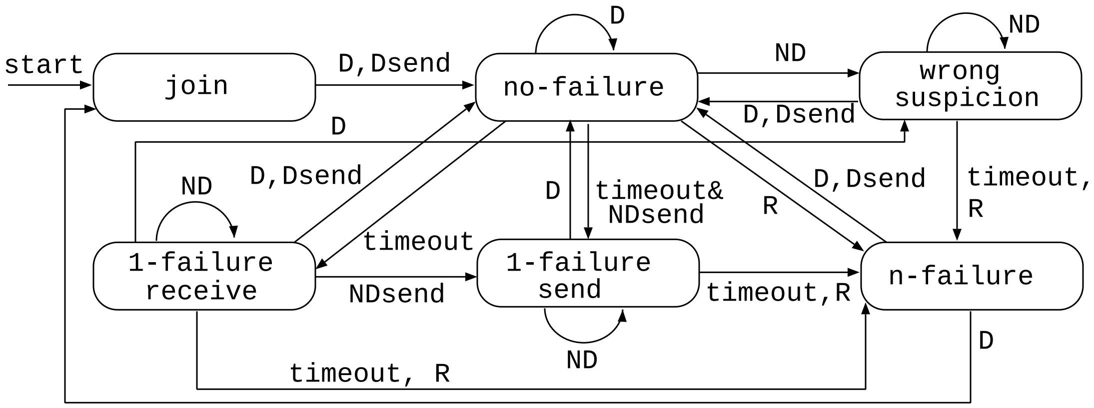

Figure 2: States of a group creator: D = decision, R = reconfiguration, and ND = no-decision message received from expected sender; *timeout* = no message received from expected sender; D_{send}, ND_{send} = group creator sends this control message.

receives p's join messages in a timely manner, it includes p in its alive-list. Let the current member q be the successor of p in the next group g (to be formed) which includes p and q. When q becomes the decider and if all group members have included p in their alive-list, q creates a new group g that includes p; recall that group members piggyback their alive-lists on all control messages they send. In addition, q retrieves its application state by calling a dedicated function provided by the application, and updates the state of p by sending this retrieved application state and undelivered proposals from its proposal buffer to p. Process p updates its application state by using the application state received from q and installing it by invoking a function provided by the application.

When the system starts, all processes are in join state. The membership protocol creates the first group as follows. Each process p in join state maintains a list of processes (*join-list*$_p$) which have sent a join message during the last N-1 slots and p has received this join message; join-list$_p$ always contains p itself. All processes include their join-list in their join messages. A process q becomes the decider when (1) join-list$_q$ contains a majority of processes, and (2) q has received from each process p in join-list$_q$ a join message which p sent in its last time slot and join-list$_q$ = join-list$_p$. Process q creates a new group containing exactly the processes in its join-list. Note that a new decider is elected as soon as a majority of the processes agrees on the join-list. In case a process p would not receive the decision message from the newly elected decider q, then p would remove q from its join-list because q does not send a join message during its time slot. Therefore, another process cannot use the same join-messages like q to become the new decider. This ensures that no two deciders can be elected which exist at the same time.

If p is in no-failure state and a timeout failure of the expected decider s occurs, it transits to either 1-failure-send or 1-failure-receive state to request that s be excluded from the membership. If p is the successor of s, it sends a no-decision message indicating that it suspects that s has failed and, after sending this message, it switches to 1-failure-send state. Otherwise, p transits to 1-failure-receive state. If p is in no-failure state and receives a no-decision message from the expected decider d, it transits to wrong-suspicion state, indicating that d has apparently missed a decision which p has received. Finally, if p is in no-failure state and receives a reconfiguration message from the expected decider, it switched to n-failure state, indicating that at least two failures have occurred.

A process p in wrong-suspicion state checks which process is suspected to have failed. If p itself is suspected, it resends its last control message after the receipt of each no-decision message. This ensures that if p's last control message was lost because of a communication failure, other group members can receive the retransmitted control message before they decide to exclude p from the membership. Notice however, that in our system model there is no guarantee that this retransmitted control message will be received by other group members. Indeed, in an asynchronous system, we cannot guarantee that a live group member will not be excluded from the membership [15]. In wrong-suspicion state a process p expects to receive a no-decision message or a decision message at least once in every D time units. In this state, if a timeout failure of the expected sender occurs or if p receives a reconfiguration message from the expected sender, it transits to n-failure state indicating that multiple failures have occurred. When process p in wrong-suspicion state receives a no-decision message from its predecessor, it creates a decision message using the information it has received from q's last decision message and switches to no-failure state, where q is the suspected process. If p receives a decision message from the expected sender and it is still a member of the current group, it transits to no-failure state. If, on the other hand, p is no longer a member of the current group, i.e. p is the suspected process, p switches to join state.

A process p in 1-failure-receive state expects to receive a no-decision or a decision message every D time units. If it receives a no-decision message from its predecessor and it is not the predecessor of the suspected process q, then p sends a no-decision message and switches to 1-failure-send state. If p receives a no-decision message from its predecessor and it is the predecessor of the suspected process q and the current membership includes more than a majority of the processes, then it removes q from the membership, switches to no-failure mode, and becomes the new decider. In this case all remaining group members have agreed that q has failed by sending a no-decision message and the newly created group contains at least a majority of the processes. If p receives a no-decision message from its predecessor, and it is the predecessor of the suspected process q, and the current group-list has exactly a majority of the processes, it sends a reconfiguration message and switches to n-failure state. This is because p is not allowed to create a group which does not contain a majority of the processes. If in 1-failure-receive state, p receives a decision message from the suspected process, it changes to wrong-suspicion state and if p receives a decision message from the expected sender, it transits to no-failure state. Finally, if a timeout failure occurs or p receives a reconfiguration message from the expected sender, p switches to n-failure state.

A process p in 1-failure-send state has already sent a no-decision message indicating that it suspects the failure of a member q. If it receives a decision message in which q is removed from the membership, it switches to no-failure state. As long as it receives a no-decision message at least every D time units, it stays in 1-failure-send state. If it does not receive a no-decision message in some D period receives a reconfiguration message from the expected sender, it switches to n-failure state.

After the role of the decider is lost, the remaining group members first try to elect a new decider using the mechanism described above by sending no-decision message. In case this mechanism fails to elect a new decider—possible when multiple failures occur—a second election mechanism is applied which is based on sending of reconfiguration messages periodically. The election algorithm has to ensure that at most one decider is created. This is complicated by the fact that when a process participates in the election of a new decider, further failures can force processes to start a new election, and this could lead to the instantiation of multiple deciders. Using synchronized clocks, we can solve this problem in a simple way: a process can only participate in one election per cycle and messages sent to elect a new decider can only used for about $(N\text{-}1)D$ time units after they were sent and by at most one process. To understand this, suppose process p participates in an election and this leads to the instantiation of a new decider and, suppose p later on participates in a new election. Since, p will no longer assume the role of decider in the group resulted from the earlier election, it will take at most N slot times (one cycle) for the role of decider to be lost in this earlier group. Hence, if a p waits until its next time slot (one cycle) before it participates in a new election, there can be at most one decider at any time.

Each process p in n-failure state sends a reconfiguration message in each of its time slots and maintains a list of processes (*reconfiguration-list*) which have sent a reconfiguration message during the last N-1 slots and p has received this reconfiguration message; p always includes itself in its reconfiguration-list. A reconfiguration message sent by p contains p's reconfiguration-list, the timestamp of the last decision message m that p has sent or received, and field *oal* of m. To create a new decision message, a process p which has sent a reconfiguration message with timestamp ts has to ensure that there exists a majority S of processes that satisfy the following properties: (1) all processes in S have sent a reconfiguration message during their last time slot, (2) the reconfiguration-list received from processes in S is identical to p's reconfiguration-list. (3) the timestamps included in the reconfiguration messages from processes in S are not greater than ts, and (4) all processes in S were in the last group p is aware of. When these four conditions are satisfied and S is maximal with respect to these conditions, then p creates a new group containing exactly the processes in S, sends a new decision message, and switches to no-failure state. Note that no two processes can use the same reconfiguration messages to become the new decider because the first process p which can use these reconfiguration messages does not send a reconfiguration message and hence all successive processes exclude p from their reconfiguration-list.

If a process q in n-failure state receives a decision message from the expected sender that includes a new membership containing q, it switches to no-failure state. Otherwise, if q is not included in the new group, and q has received a decision message of all new group members, it transits to join state. This delayed switch to the join mode ensures that when the role of the decider is lost in less than a round, q could participate in a new election; otherwise, when all new group member have send a decision message, q knows that it is not in their last group and hence won't be in any new group created by these processes. In our failure assumptions, we assumed that at least a majority of processes S which were members of the last group survive until a new process is reintegrated into the system. This ensures that the a new decider is elected as soon as all processes in S can communicate in a timely manner.

A no-decision message and followed by a reconfiguration message sent by the same process p could result in multiple deciders, because both election could be successful. To avoid this situation, when p switches to n-failure state, it does not participate in a new election for the duration of N-1 slot times. When the first election is successful and p becomes the decider it transits to no-failure mode; otherwise, the election was not successful or the decider role created by previous election is lost in at most (N-1)D time units, because p does not assume the decider role created by the previous election. During its waiting period p transmits a reconfiguration message with an empty reconfiguration-list in its time slot. This ensures that p cannot be participate in an election for at least N-1 slots and that a new decider is typically elected in two rounds.

5.4 Removal of Updates

Because a process q can be removed from the membership before all proposals of q are stable, it is possible that some proposals whose proposal descriptors are included in the *oal* have not been received by any member of the current group. Clearly such updates cannot be delivered, hence cannot become stable, and must therefore be *removed* from *oal*. The problem is to ensure that all current group members deliver an update whose proposal descriptor is not removed from *oal*, and no group member delivers an update whose proposal descriptor is removed from *oal*.

To remove such proposals from *oal*, each member p sends its current view (v_p) of the *oal* in all no-decision/reconfiguration messages it sends. Process p's view v_p is derived from field *oal* from the decision message m with the highest send timestamp that p has received: p uses field *oal* from m and updates the acknowledgment bits. Each process p also includes a field *dpd* (delivered proposal descriptors) in all no-decision/reconfiguration messages it sends; this field contains a list of all proposal descriptors p has delivered, but which have an undefined ordinal so far. Field *dpd* will be used by the new decider to append all proposals to the newly created *oal* which have undefined ordinals, but which have already been delivered by a new group member. This is necessary to ensures the atomicity constaints. View v_p together with field *dpd* contains acknowledgements for all unstable proposals p has received so far. Suppose, p sends a no-decision/reconfiguration message requesting that a process q be removed from the membership. In this case, p marks a proposal pr as *undeliverable* in its proposal descriptor buffer if pr was proposed by q and p hasn't received pr so far. In addition, p marks all those proposals undeliverable that are proposed by q and are received after p has sent the no-decision/reconfiguration message. A process does not deliver or acknowledge any proposal that is marked as undeliverable. An undeliverable mark on a proposal is automatically cleared after one cycle, unless it was set again. This is because a no-decision/reconfiguration message can only be used for the creation of a new group for at most the duration of $N - 1$ slots.

Let us assume that a new group is created by a process r by removing process q from the membership. Since every group member includes its current view of the *oal* in no-decision/reconfiguration messages, process r has the current view of the *oal* from all new group members. It uses these views to update the acks in its proposal descriptor buffer and the acks in its current view of the *oal*. The election of r guarantees that each view which r has received is a prefix[2] of r's current view of the *oal*, because r includes in the new group only those processes that were members of the last group and has the current view of the *oal* with the highest send timestamp. Decider r uses its current view of the *oal* as new *oal*, but it marks some proposal descriptors in *oal* as *removed* to indicate that no group member will deliver the corresponding update. The proposal descriptors marked as removed are deleted from *oal* by a decider when these descriptors reach the head of *oal*. Furthermore, each group member purges all proposals marked as removed from their *pdb* and *pb*. A proposal descriptor of a proposal pr proposed by a process q is marked as removed in *oal* by decider r, if any of the following conditions are satisfied: (1) this proposal descriptor is not acknowledged by any member of the new group, or (2) pr is proposed with total/time order delivery semantics and there exists a proposal ppr proposed by q whose ordinal is smaller than that of pr and the proposal descriptor of ppr has been marked as removed, or (3) pr is proposed with a strong/strict atomicity semantics and there exists an earlier proposal ppr proposed by q or any other process which is removed from the membership whose ordinal is smaller than or equal to the *hdo* of pr and the proposal descriptor of ppr has been marked as removed.

The rationale behind condition (1) is that no current group member has received pr, and so the update proposed in pr cannot be delivered. The rationale behind condition (2) is that both the total and the time order delivery semantics must preserve the FIFO property, i.e the updates proposed by the same process must be delivered in the order they are proposed. If condition (2) is satisfied, the update proposed in ppr is not delivered and so, the update proposed later by the same sender in pr should also be not delivered. Finally, the rationale behind condition (3) is that strong/strict atomicity requires that the update proposed in pr can be delivered by a member p only after p or a majority has received all earlier proposals on which pr could depend, i.e. proposals with ordinals not greater than the *hdo* of pr. This ensures that all these earlier proposals will eventually be delivered by all group members or p will be removed from the membership. If condition (3) is satisfied, the update proposed in ppr will not be delivered by any new group member, and so the update proposed in pr must not be delivered. Note that no proposal of a new group member can depend on a removed proposal, because otherwise this new group member or majority has received this proposal and it would therefore not be removed.

Some proposals with strong/strict atomicity cannot be delivered even when they are not affected by the above rules because they are not ordered so far and which were proposed by a process q which is removed from the membership. Consider the following situation: process q has sent proposal pr with strong/strict atomicity; before sending pr process q was the decider and no member of the new group has received its decision message in which p has ordered some proposals so that the *hdo*

[2] We hereby ignore the purging of stable descriptors and different values in the acknowledgment fields.

of pr is greater than the highest ordinal known to the remaining group members. These processes cannot deliver pr because they do not know the proposal referred by hdo. Furthermore, pr's ordinal has to be higher than the ordinals of the proposals it depends on and hence it cannot be ordered in the views of the oal from the members of the new group. When process q is removed from the membership, then also all proposals of q which haven't been ordered and delivered so far are deleted by the above described deletion rules, i.e. pr is not delivered by any new group member. Proposals which have already been delivered by some members, but which haven't been ordered so far, are appended to the oal by the new decider using field dpd of the reconfiguration/no-decision messages received from all new group members.

6 Discussion

Our objectives in designing Timewheel protocols were to support multiple semantics concurrently and provide good overall performance. The unique features of this protocol are as follows.

1. It supports nine different group communication semantics ranging from very weak to very strong semantics, and a user can dynamically select appropriate semantics when broadcasting an update. As a result, users can tailor their applications according to the application requirements by choosing weaker semantics whenever possible.
2. It provides time order semantics which preserves causal order arising due to hidden channels and ensures total order between updates delivered to different teams.
3. It uses an internal clock synchronization service, which aids in supporting different semantics and simplifies the design of broadcast and membership protocols.
4. It provides good overall performance in the absence as well as presence of communication and process failures.

Several asynchronous group communication protocols have been proposed in literature [5, 20, 3, 19, 1]. The Timewheel group communication protocol is different from all these protocols in supporting several different semantics and allowing the applications to choose the appropriate semantics dynamically when broadcasting an update. For example, in [1], a user is limited to using total order and weak atomicity, while [5] provides total order/strict atomicity (when constant $L > N/2$). The protocols proposed in [3] and [19] provide causal order (that does not preserve hidden causality) and total order. However, a user has to specify the appropriate order semantics in advance.

The delivery time provided by the Timewheel reliable broadcast protocol in the absence of failures is as low as that provided by the fixed sequencer-based broadcast protocols such as [3, 16, 4]. Unlike these protocols, the Timewheel protocol rotates the decider role to ensure balanced load and small stability time in the presence of fast as well as slow update arrival rates. The flow control techniques further improve the performance over these fixed sequencer-based protocols when updates arrive at a very fast rate or when the frequency of communication failures increases. The Timewheel protocol differs from other rotating decider protocols like [5, 13] in its use of positive acknowledgements instead of negative acknowledgments to simplify and to speed up the exclusion/inclusion of processes from/to the membership and the establishment of message stability. By batching multiple acknowledgements into one message the Timewheel protocol can achieve the number of messages exchanged per update at least as low as in negative acknowledgment schemes.

The stability time provided by the Timewheel reliable broadcast protocol is as low as that in token-based protocols [8, 1]. However, in the token-based protocols, a group member has to wait until it receives the token to broadcast an update, while in the Timewheel protocol, an update is broadcast as soon as it is received from the application. This results in a much smaller delivery time in the Timewheel protocol, and also improves the stability time.

The Timewheel membership protocol is different from other asynchronous membership protocols such as [21, 18, 1] in that it makes use of an internal clock synchronization service to detect process failures, elect a single decider in the system, and provide desired atomicity semantics. The use of clock synchronization service results in a much simpler design. In addition, the Timewheel membership protocol minimizes interference with the broadcast protocol, minimizes the possibility of falsely detecting a failure, and is optimized for single failure scenarios, which are more likely to occur than the multiple failure scenarios.

7 Conclusions

We proposed a membership and a reliable broadcast protocol: the Timewheel group communication protocol. This protocol allows clients to use different semantics simultaneously and optimize performance in case weaker semantics are sufficient. The broadcast protocol therefore supports very weak and very strong semantics. A client can select time order/strict atomicity semantics to guarantee causal delivery in case processes can communicate by hidden channels. When communication by hidden channels can be excluded, a client can use total order/majority atomicity semantics to ensure causal delivery. To get the best delivery times a client can choose unordered/weak atomicity semantics.

The broadcast protocol uses a rotating decider scheme to balance the processing and communication load between all group members and to detect process failures. Instead of using a negative acknowledgment scheme, the broadcast protocol employs positive acknowledgments to simplify the integration of new processes and the removal of failed processes from the membership. However, it does not need more messages per update than negative acknowledgment schemes because it allows to acknowledge multiple proposals with one acknowledgement message. To reduce the number of messages, the clock synchronization, the membership, and the broadcast protocol can piggy-back information on all control messages. The membership protocol masks a single crash failures or a loss of a decision message with a simple one round election protocol. Multiple failures are masked using a time-slotted election protocol.

References

[1] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of Thirteenth International Conference on Distributed Computing Systems*, pages 551–560, Pittsburgh, PA, May 1993.

[2] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb 1987.

[3] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.

[4] R. Carr. The Tandem global update protocol. *Tandem Systems Review*, Jun 1985.

[5] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.

[6] D. R. Cheriton. Understanding the limitations of causally and totally ordered communication. In *Proceedings of the Forteenth ACM Symposium on Operating Systems Principles*, pages 44–57, Asheville, NC, 1993.

[7] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989.

[8] F. Cristian. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin*, 33(9):115–116, Feb 1991. Also presented at the First IEEE Workshop on Management of Replicated Data, Houston, TX, (Nov 1990).

[9] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.

[10] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in the Advanced Automation System. In *Proceedings of the Twentieth Symposium on Fault-Tolerant Computing*, pages 6–17, Newcastle-upon-Tyne, UK, Jun 1990.

[11] F. Cristian, R. de Beijer, and S. Mishra. A performance comparison of asynchronous atomic broadcast protocols. *Distributed Systems Engineering*, 1(4):177–201, Jun 1994.

[12] F. Cristian and C. Fetzer. Fault-tolerant internal clock synchronization. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*, Dana Point, Ca., Oct 1994.

[13] F. Cristian and S. Mishra. The pinwheel asynchronous atomic broadcast protocols. In *Proceedings of the Second International Symposium on Autonomous Decentralized Systems*, Phoenix, AZ, Mar 1995. to appear.

[14] C. Fetzer and F. Cristian. Semantics of an asynchronous group-communication services. Technical Report CS95-XXX, Dept of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, 1995.

[15] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[16] M. F. Kaashoek, A. Tanenbaum, S. F. Hummel, and H. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, Oct 1989.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, Jul 1978.

[18] S. Mishra, L. Peterson, and R. Schlichting. A membership protocol based on partial order. In J. F. Meyer and R. D. Schlichting, editors, *Dependable Computing for Critical Applications 2*, pages 309–331. Springer-Verlag, Wien, 1992.

[19] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal*, 1993. *to appear.*

[20] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug 1989.

[21] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Eleventh ACM Symposium on Principles of Distributed Computing*, Montreal, Canada, Aug 1991.