The Timewheel Group Membership Protocol*

Shivakant Mishra
Department of Computer Science
University of Wyoming
Laramie, WY 82071-3682, USA
http://www.cs.uwyo.edu/~mishra

Christof Fetzer and Flaviu Cristian
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093−0114, USA
http://www-cse.ucsd.edu/~{cfetzer|flaviu}

Abstract. We describe a group membership protocol, called the timewheel group membership protocol, for a timed asynchronous distributed system. This protocol is a part of the timewheel group communication service that supports multiple group communication semantics simultaneously. The timewheel group membership protocol is unique in several respects. First, it has been designed for a *timed* asynchronous distributed system model. Second, it is optimized for those failure scenarios that are more likely to occur than others. In particular, it uses a very simple and fast algorithm to recover from single failures. Furthermore, the group communication service is not interrupted, if a failure suspicion turns out to be a false alarm. Third, this protocol incurs minimal processing load during failure-free periods. In fact, this protocol does not cause any extra messages to be exchanged during failure-free periods. Finally, as a consequence of using the timed asynchronous distributed system model, this is one of the first few non-real-time membership protocols that are *timed*, i.e. its specification describes what outputs and state transitions occur in response to inputs and the time it takes these outputs and state transitions to occur.

1 Introduction

With the ever increasing use of computers in everyday life, particularly in critical applications, the need for high performance, dependable systems is increasing. Unfortunately, building high performance, dependable systems is complicated. One technique to construct a dependable service is to implement it by a *team* of replicated servers. The underlying idea is that the currently running team members, i.e the current *group* of servers that implement the service, maintain a *consistent* replicated service state and, if one 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 or failures [7].

A *group membership protocol* is a part of a group communication service that is useful in maintaining a consistent replicated service state in the presence

* Supported in part by AFOSR grants F49620-91-1-0103 and F49620-98-1-0070, and a faculty grant-in-aid from the Office of Research, University of Wyoming.

of communication or processor failures [9]. Informally, it provides a consistent system-wide view of which team members are operational at any given moment in time.

In this paper, we present a group membership protocol called the *timewheel* group membership protocol. This protocol is a part of the timewheel group communication system [17, 19] that supports multiple group communication semantics simultaneously and provides excellent overall performance in the absence as well as presence of communication or process failures. In particular, the timewheel group communication service provides three kinds of ordering semantics—*unordered*, *total ordered* and *time ordered*, three different kinds of atomicity semantics—*weak*, *strong* and *strict atomicity*, and one termination semantic.

The timewheel membership protocol is unique in several respects. First, it has been designed for a *timed* asynchronous distributed system model [10]. This model has been proposed recently. Most distributed systems based on non-real-time operating systems and communication services, such as Unix and UDP, are timed asynchronous. Second, while this protocol deals with all failure scenarios within the failure model assumed, it is optimized for those scenarios that are more likely to occur than others. In particular, it uses a very simple and fast algorithm to recover from single failures. Furthermore, the group communication service is not interrupted, if a failure suspicion turns out to be a false alarm. Third, this protocol incurs minimal processing load during failure-free periods. In fact, this protocol does not cause any extra messages to be exchanged during failure-free periods. Finally, as a consequence of using the timed asynchronous distributed system model, this is one of the first few asynchronous membership protocols that are *timed*, i.e. its specification describes what outputs and state transitions occur in response to inputs and the time it takes these outputs and state transitions to occur. Timed asynchronous membership protocols proposed earlier are described in [13, 8].

2 System Model

We assume a timed asynchronous distributed system [10]: the system consists of a finite set of processes linked by an asynchronous datagram service. A one-way time-out delay δ is defined for the datagram service. Although there is no guarantee that a message will be delivered within δ time units, a message is *likely* to be delivered within δ. We say that a process receives a message m in a *timely manner*, if the transmission delay of m is not greater than δ. When the transmission delay of m is greater than δ, we say that m has suffered a performance failure or that m is *late*. The asynchronous datagram service has omission/performance failure semantics [7].

The process management service defines a maximum scheduling delay σ, meaning that a process is likely to react to any trigger event (such as a timer event) within σ time units. If a process p takes more than σ time units to react to a trigger event, it suffers a performance failure. When p's reaction time is at most

σ, we say that p is *timely*. We assume that processes have crash/performance failure semantics [7].

Each process p has access to a local hardware clock H_p. The drift rate of a correct hardware clock is bounded by an a priori given constant ρ. Hardware clocks are not synchronized: the deviation between two correct hardware clocks can be arbitrarily large. For most quartz clocks available in modern computers, the maximum hardware clock drift rate ρ is of the order of 10^{-4} to 10^{-6}. We assume that hardware clocks have *crash* failure semantics and that a non-crashed process has a correct hardware clock.

This system model is significantly different from the well-known *time-free* asynchronous system model [18], in which services are time-free, i.e. 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. Because in this 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 [18, 23, 3]. Most existing distributed systems based on non-real-time operating systems and communication services, such as Unix and UDP, are timed asynchronous.

The timewheel membership protocol described in this paper is a part of the timewheel group communication service that also includes a clock synchronization protocol and an atomic broadcast protocol. The clock synchronization protocol keeps the local clocks of correct processes *synchronized*: (1) the deviation between two synchronized clocks is bounded by a given constant Ψ, and (2) the drift rate of the synchronized clocks is within a known linear envelope of real-time. In the timed asynchronous system model, it is not possible to keep correct clocks synchronized all the time, because it allows a (very unlikely) run in which no process can communicate with any other process. We therefore use a *fail-aware* clock synchronization protocol [15] that guarantees that (1) any process p knows at any time if its clock is synchronized, and (2) whenever the underlying datagram and process service allows this, p's clock is synchronized. A process p that cannot keep its clock synchronized is removed from the current group by the group membership protocol. When p can synchronize its clock again, p applies to join the group again.

The timewheel broadcast protocol [19] is an optimization and extension of the protocols proposed in [5] and [11, 12]. A broadcast of an update may be initiated by a member at any time by sending a *proposal message* to all group members. Another type of message called a *decision message* is used to associate unique numbers, called ordinals, to updates/membership changes being broadcast[2], establish the stability of broadcast updates, and to detect message losses. A group member, called the *decider*, is responsible for sending decision messages. A decision message includes an *ordering and acknowledgement list* (referred to as *oal* henceforth) consisting of update/membership change descriptors, along with in-

[2] The delivery order of updates/membership changes is not necessarily same as the order of the ordinals associated with them.

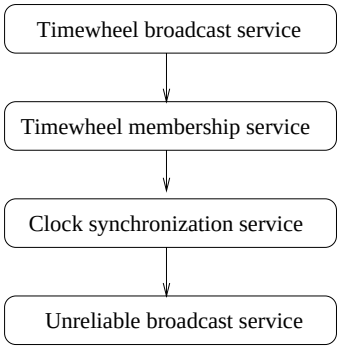

Fig. 1. System Architecture of the Timewheel Group Communication Service.

formation about which group members have received those update/membership changes.

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. All group members are cyclically ordered. A group member d relinquishes its decider role by sending a decision message in at most D time units, and the next group member in the cyclical order assumes the decider role on receiving this decision message. Each member maintains two buffers—a *proposal buffer*, to store the received proposals, and a *proposal descriptor buffer*, to store proposal descriptors and their ordinals. Both of these buffers are updated on receipt of proposal or decision messages. Updates stored in these buffers are delivered to the clients when three delivery conditions, *atomicity*, *order*, and *general* (see [19] for details), are satisfied.

3 Membership Specification

The timewheel membership protocol is a majority agreement protocol [8]: (1) it provides a sequence of *completed* majority groups, where a completed majority group is a majority group joined by all its members, and (2) all members of a completed majority group g in this sequence agree on a history of replica updates when they join g. A consequence of the majority agreement membership protocol is that there may be some limited divergences between the histories seen by the members of completed majority groups and other team members [8].

A *team* is a set of processes. In this paper, we are only interested in the team of processes executing the timewheel group communication service, and so, we will use the term *process* to refer to a process in that team. The timewheel group membership protocol maintains a consistent system-wide current group (sometimes also called "view") of processes that exhibit "synchronous behavior". The meaning of synchronous behavior is formalized by predicate Δ-stable [15]. A process p is Δ-stable iff (1) p is timely, (2) p can communicate in a timely manner

with a majority of processes and all these processes are Δ-stable, and (3) p can detect all messages from non Δ-stable processes as being late and, therefore, can reject them. The membership protocol tries to provide each process with an *up-to-date* group of processes that currently exhibit synchronous behavior. Because processes can be partitioned, it is not always possible that all processes maintain an up-to-date group. The timewheel membership protocol is *fail-aware* in the sense that a process knows at any point in time if its current group is up-to-date.

More precisely, the membership protocol satisfies the following properties (see [16]): (1) If a process is Δ-stable for at least τ time units, it has an up-to-date group, (2) at any point T in clock time, if p and q have an up-to-date group at T, their group is identical, (3) if a process is Δ-stable for at least τ time units, it is included in any up-to-date group, (4) if a process' current group has been out-of-date for τ time units, it is excluded from all up-to-date groups, and (5) an up-to-date group contains at least a majority of the processes.

The group membership problem or the atomic broadcast problem is not solvable in the time-free asynchronous system model [18, 4]. However, existing asynchronous systems have typically enough "synchronism" to allow a deterministic solution of the group membership or the atomic broadcast problem. For example, a typical execution of a system consists of long periods in which the system is Δ-stable interleaved by relatively short periods in which the system is not Δ-stable. We formalize this observation by a *progress assumption* [14]: we assume that the system will be infinitely often Δ-stable. This progress assumption allows a deterministic solution of consensus [14]. Since the consensus problem is as hard as the group membership or the atomic broadcast problem [4], it also allows a deterministic solution of the group membership or the atomic broadcast problem.

4 Protocol Description

4.1 Overview

This protocol is based on the ideas developed in the membership protocols described in [20, 1, 21, 2]. Informally, the key idea in these protocols is that the team members exchange membership messages and each live team member p maintains a set $\mathcal{S}_p$ of live team members based on these messages. The team members continue to exchange membership messages until the following property is satisfied: $\forall q \in \mathcal{S}_p, \mathcal{S}_q = \mathcal{S}_p$.

We conceptionally split the timewheel membership protocol into two parts at each team member—a *failure detector* and a *group creator*. Each failure detector maintains an *alive-list* of team members that are currently functioning correctly. A failure detector is unreliable, i.e. an alive-list can contain team members that have failed, or there might exist some team members that are live but not in the alive-list. Furthermore, the alive-lists maintained by different failure detectors can be different. A *group creator* uses this alive-list to maintain a *group-list*. It

guarantees that all correct team members in a group-list agree on the current and past group-lists. We call a process in a 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 send 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 occurs, a *multiple-failure* election protocol is started to create a new group.

This protocol uses three control messages: *no-decision*, *join*, and *reconfiguration*. In addition, a *decision* message, which is a part of the timewheel atomic broadcast protocol, is also treated as a control message by the membership protocol. A failure detector keeps all group members under surveillance by checking that they send control messages periodically. Recall that group members take turns to take the role of decider in the timewheel atomic broadcast protocol, and a decider sends a decision message in at most D time units after it takes the decider role. So, a failure detector expects to receive a decision message after at most D time units from the current decider d, a decision message from the successor of d after at most D time units after receiving the decision message from d, and so on.

A failure detector suspects that a group member p has failed, if it doesn't receive a control message from p in the expected interval of time. In such a case, it informs the group creator that a member 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. This failure detection algorithm is similar to an *attendance list* or a *neighbor surveillance* protocol that has been proven to require the minimum number of messages that must be exchanged to detect member failures [6].

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 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. The global time-base provided by the synchronized clocks is divided into cycles and the cycles are divided into slots; each team member has exactly one slot per cycle. Each team member p sends a *reconfiguration* message during its time slot that 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.

After the role of the decider is lost, the remaining group members first try to elect a new decider using the single-failure election protocol. 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. We solve this problem by using synchronized clocks in a simple way: a process can only participate in one election per cycle and messages sent to elect a new decider can only be 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 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.

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.

4.2 Detailed Description

Failure Detector Let N denote the total number of team members and FD_p denote the failure detector of process p. The alive-list of FD_p contains p and each process q, such that p has received at least one *control message* from q in the last N slots. A failure detector uses the send timestamps to maintain its alive-list. The length of each time slot has to be at least $D + \delta$, where D is the maximum time interval after which a decider sends a decision message, and δ is the one-way timeout delay described in Section 2.

During a failure free period, all group members are in the alive-list, i.e. no group member is suspected to have failed. A failure detector 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 $ts+2D$. In the following discussion, we call e the *expected* sender. 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. In the following discussion, we say that a *timeout* failure has occurred, when a failure detector informs a failure suspicion. 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.

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. Group creators of other members use this descriptor to update their group-lists. We describe a group creator as a finite state machine with six states (see Figure 2): *join*, *failure-free*, *wrong-suspicion*, *1-failure-receive*, *1-failure-send*, and *n-failure*.

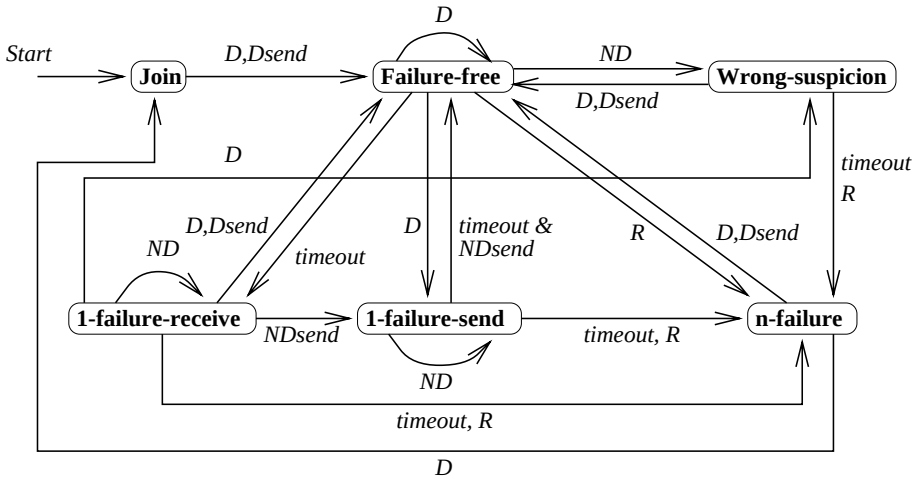

Fig. 2. State Transition Diagram of a Group Creator: D = decision, R = reconfiguration, and ND = no-decision message received from expected sender; $timeout$ = no message received from expected sender; D_{send}, ND_{send} = group creator sends this control message.

The join state handles the creation of a new group. This new group consists of the members of the current group and processes which want to be integrated into

the group. During stable periods, when there are no communication or process failures, all group members are in failure-free state. They all become decider periodically, and send at least one decision message per cycle. The 1-failure-receive, 1-failure-send, and the wrong-suspicion states are 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. A process enters a 1-failure-receive or 1-failure-send state, if it concurs with a single failure suspicion. A process enters a wrong-suspicion state, if it does not concur with a single failure suspicion. This 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. Finally, the n-failure state is used to handle more than one failures during a cycle.

Join State

All team members are in join state at the start of the system. In addition, a team member is in join state after it recovers from a crash failure. The membership protocol creates the first group as follows. Each process p in join state maintains a list, *join-list*$_p$, of all those processes from which p has received at least one join message in the last N-1 slots; join-list$_p$ always contains p itself. A process includes its join-list in each join message it sends. A process q becomes the decider in its time slot, when (1) join-list$_q$ contains a majority of the processes, and (2) q has received a join message from each process p in join-list$_q$ in the p's last time slot, such that join-list$_q$ = join-list$_p$. Process q creates a new group containing exactly the processes in its join-list, transits to a failure-free state, and sends a decision message (the first decision message). This decision message includes a membership descriptor containing the membership of the new group.

This algorithm guarantees that more than one deciders cannot be elected at the same time. This is because a new decider is elected as soon as a majority of the processes agrees on the join-list. In case, another process p does not receive the first decision message from the newly elected decider q, p will remove q from its join-list. So, another process cannot use the same set of join messages as used by q to become a decider. On receiving a decision message, a team member p transits to the failure-free state, if it is included in the membership of the new group.

If a group has already been created, a process p in the join state joins the group as follows. 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 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.

Failure-free State

If p is in failure-free state and a timeout failure of the expected decider s occurs, it transits to either 1-failure-send or 1-failure-receive state. These states indicate that a single communication failure (loss of a decision message) or a single process crash (failure of s) has occurred. 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 failure-free 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 failure-free state and receives a reconfiguration message from the expected decider, it switches to n-failure state, indicating that at least two failures have occurred.

Wrong-suspicion State

A process p is in wrong-suspicion state when a single failure has been suspected and p does not concur with this suspicion. In this state, p 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 transient communication failure, other group members are expected to receive this retransmitted control message before they decide to exclude p from the membership. However, notice that in our system model, there is no guarantee that this retransmitted control message will be received by other group members. Indeed, in a timed asynchronous system, we cannot guarantee that a live group member will not be excluded from the membership.

In a 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 from the expected senders. 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. If p in wrong-suspicion state receives a no-decision message from its predecessor, it assumes the role of the decider and switches to failure-free state. In the failure-free state, p will create a decision message using the information it has received from q's last decision message, where q is the suspected process, and send this decision message as usual. If p in wrong-suspicion state receives a decision message from the expected sender and it is still a member of the current group, it transits to failure-free state. This situation occurs when another group member that does not concur with the failure suspicion has sent a decision message. If, on the other hand, p receives a decision message from the expected sender, and it is no longer

a member of the current group, i.e. a new group has been formed that does not contain p as a member, p switches to join state.

1-failure-receive State

A process p in 1-failure-receive state expects to receive a no-decision or a decision message every D time units from the expected sender. Let q be the suspected process. If p receives a no-decision message from its predecessor and p is not q's predecessor, 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 p is q's predecessor, then all members of the current majority group except q have agreed that q has failed by sending a no-decision message. In this case, p can form a new group if there are a majority of processes still remaining that concur with q's failure. So, if the current membership includes *more* than a majority of the processes, then p removes q from the membership, switches to failure-free mode, and becomes the new decider. If, on the other hand, the current group-list has *exactly* a majority of the processes, p is not allowed to create a new group. In this case, it sends a reconfiguration message and switches to n-failure state.

If, in 1-failure-receive state, p receives a decision message from the suspected process, it transits to wrong-suspicion state. If p receives a decision message from the expected sender, it transits to failure-free state. Finally, if a timeout failure occurs, or if p receives a reconfiguration message from the expected sender, p switches to n-failure state.

1-failure-send 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. While in 1-failure-send state, if a timeout failure occurs at p, or if p receives a reconfiguration message from the expected sender, it switches to n-failure state. If a no-decision message is received from the expected sender, p stays in 1-failure-send state. Finally, if a decision message is received from the expected sender, p switches to failure-free state.

n-failure State

A process p in n-failure state maintains a list of processes, called the *reconfiguration-list*. This list contains p and all those processes from which p has received a reconfiguration message in the last N-1 slots. p send a reconfiguration message in each of its time slots. A reconfiguration message sent by p contains p's reconfiguration-list, the timestamp ts of the last decision message m that p knows about, and field oal of m. A process p that has sent a reconfiguration message with timestamp ts, creates a new group during its time slot, if there exists a majority $\mathcal{S}$ of processes such that the following properties are satisfied: (1) p has received a reconfiguration message from all processes in S in their last time slot, (2) the

reconfiguration-list received from all processes in S in the last time slot is identical to p's reconfiguration-list, (3) timestamps included in the reconfiguration messages from processes in S in the last time slot are not greater than ts, and (4) all processes in S were in the last group p is aware of. The new group created by p contains exactly the processes in S and p is the decider of the new group. p sends a new decision message and switches to failure-free state. Notice that no two processes can use the same set of 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 they do not receive p's decision message.

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 failure-free state. If q is not included in the new group, it waits until it has received a decision message from all new group members. After receiving decision messages from all new group members q switches to join state. This delayed switch to the join state ensures that when the role of the decider is lost in less than a decider round, q could participate in a new election. 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.

Notice that in the protocol described so far, a no-decision message followed by a reconfiguration message sent by the same process p could result in multiple deciders. This is because both the single-failure and multiple failure elections could be successful in this case. 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. If the first election is successful and p becomes the decider, it transits to failure-free state. If the single-failure election is successful, but p does not assume the role of the decider, the decider role created by the single-failure election will lost in at most $(N\text{-}1)D$ time units. During this waiting period, p transmits a reconfiguration message with an empty reconfiguration-list in its time slot. This ensures that p will not participate in an election for at least N-1 slots and that a new decider is typically elected in two rounds.

4.3 Preserving Order and Atomicity Semantics

Whenever a change in the group membership occurs due to process departures, some of the updates proposed by the departed group member(s) may have to be discarded. This is needed to ensure various ordering and atomicity semantics that the timewheel group communication service provides. Hence some proposal descriptors have to be removed from the *oal*. The key problem is to ensure that all current group members deliver an update whose proposal descriptor is not removed from *oal*, and no current group member deliver an update whose proposal descriptor is removed from *oal*.

Undeliverable Proposals

We call a proposal that should not be delivered by any of the current group members an *undeliverable proposal*. A proposal pr proposed by a departed team member q is an undeliverable proposal if it belongs to one of the following four categories: (1) *Lost proposal*: a proposal descriptor of pr is included in the oal, but no current member has received pr, (2) *orphan-order proposal*: pr is proposed with a total or time order semantics and there exists an undeliverable proposal ppr, whose ordinal is smaller than that of pr, (3) *orphan-atomicity proposal*: pr is proposed with a strong or strict atomicity semantics, and there exists an undeliverable proposal ppr, whose ordinal is smaller than or equal to the hdo of pr, or (4) *unknown dependency proposal*: pr is proposed with a strong or strict atomicity semantics, and hdo of pr is greater than the highest ordinal known to the remaining group members.

The rationale behind lost proposals is that no current group member has received pr, and so the update proposed in pr cannot be delivered. The rationale behind orphan-order proposal 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 pr is an orphan-order proposal, 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. The rationale behind orphan-atomicity proposal is that strong and strict atomicity require that the update proposed in pr can be delivered by a member p, only after p or a majority has received all proposals on which pr could depend; recall that pr can depend on all proposals with ordinals less than or equal to the hdo of pr. If pr is an orphan-atomicity proposal, the update proposed in ppr is not delivered and so, the update proposed in pr that could depend on the update proposed in ppr should also not be delivered.

Finally, the rationale behind unknown dependency proposal is that while pr has been received by some or all group members, no member knows some of the proposals on which pr could depend. This situation can occur as follows. Process q is the decider. It orders some new proposals in the oal and sends a decision message dm . After sending dm, it sends a proposal pr with strong or strict atomicity. Due to transient communication failures, some or all current group members receive pr, but none receives dm. Since pr is sent with strong or strict atomicity semantics, it can be delivered only after all proposals with ordinals smaller than or equal to the hdo of pr have been delivered. However, since q ordered some new proposals in dm and no current group member received dm, the hdo of pr is greater than the highest ordinal known to any of the current group members. So, these members do not know which new proposals were ordered by q, and hence, cannot deliver pr.

Removal of Undeliverable Proposals

The proposal descriptors of all undeliverable proposals must be removed from the oal and all such proposals must be purged from current group members'

local buffers. To remove such proposals from *oal*, each member p sends its current view (v_p) of the *oal* in all no-decision or reconfiguration messages it sends. Process p's view v_p is derived from the *oal* of the decision message m with the highest send timestamp that p has received: p uses this *oal* from m and updates the acknowledgment bits. Each process p also includes a field *dpd* (delivered proposal descriptors) in all no-decision or 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 constraints. View v_p together with field *dpd* contains acknowledgements for all unstable proposals p has received so far. Suppose, p sends a no-decision or 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 or 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 or 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 or 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[3] 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 proposal descriptors of all undeliverable proposals in *oal* as *undeliverable* to indicate that no group member will deliver the corresponding update. Furthermore, proposals which have already been delivered by some members, but which haven't been ordered so far, are appended to the *oal* by r using field *dpd* of the reconfiguration or no-decision messages received from all new group members. The proposal descriptors marked as undeliverable are deleted from *oal* by a decider when these descriptors reach the head of *oal*. In addition, each group member purges all proposals marked as undeliverable from their *pdb* and *pb*.

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

5 Implementation

The timewheel group membership protocol is currently being implemented on a network of SGI workstations (Indys) connected by a moderately loaded 10Mb/s Ethernet. This implementation uses the UDP broadcast socket interface of the Unix operating system. The single-failure election protocol has already been implemented. This implementation has been nontrivial for two reasons. First, unlike other group communication services, the timewheel group communication service provides multiple semantics concurrently. The three atomicity semantics of an update broadcast using the timewheel group communication service are affected by the group membership protocol. Hence, special care has to be taken to ensure that updates are delivered correctly in the presence of process failures and recoveries. Second, unlike in other group communication services proposed for an asynchronous distributed system, the number of events that may occur concurrently at a group member is rather large in the timewheel group communication service. This requires an efficient mechanism to handle concurrent events.

There are two common techniques to implement concurrent, event-driven software: *thread-based* and *event-based*. In the thread-based technique, a separate thread is spawned for each event type, say e_{type}, in the program. This thread waits for an event of type e_{type} to occur and takes appropriate actions on the occurrence of that event. In the event-based technique, a single-threaded event loop performs event demultiplexing and event handler dispatching in response to the occurrence of multiple events.

An initial thread-based implementation indicated that there is significant performance overhead associated with using threads. This significant performance overhead results from having to use a large number of threads and to schedule these threads explicitly. Since the number of different types of concurrent events in the timewheel group communication service is large, a large number of threads needs to be created and maintained. As a result, the performance overhead associated with creating and maintaining this large number of threads is large. Furthermore, to avoid dealing with race conditions among these threads, we schedule these threads explicitly in the protocol code. Due to a large number of threads, this takes up significant amount of time. A detailed comparison between the two techniques, thread-based and event-based, to implement group communication service is given in [22].

We chose an event-based implementation of the timewheel membership protocol. To do so, we first implemented an *event handler* that allows a client to wait for multiple concurrent events: the client can define for each event a procedure that processes that event. As soon as an event occurs, the event handler calls the appropriate procedure to allow the client to process that event. At any time, at most one event is processed and therefore no explicit synchronization between procedures that are processing the events is required. The event handler is implemented by a single thread of control and allows an efficient processing of a large number of different types of events.

6 Conclusions

We have presented a group membership protocol, called the timewheel group membership protocol, for a timed asynchronous distributed system model, which has been shown to be more general than the asynchronous distributed system model used in the past. A consequence of using the timed asynchronous distributed system model is that the protocol specification describes not only what outputs and state transitions occur in response to inputs, but also the time it takes these outputs and state transitions to occur. While this protocol deals with all failure scenarios within the failure model assumed, it is optimized for those scenarios that are more likely to occur than others. To deal with single failure scenarios, it uses a very simple and fast algorithm. It does not interfere with the other protocols, such as atomic broadcast or clock synchronization, if a failure suspicion turns out to be a false alarm. Furthermore, it minimizes the processing overhead during failure-free periods.

References

1. Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication subsystem for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, MA, Jul 1992.
2. Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The totem single-ring ordering and membership protocol. *ACM Transactions on Computing Systems*, 13(4):311–342, 1995.
3. T. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, pages 322–330, May 1996.
4. T. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 325–340, Aug 1991.
5. J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.
6. F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.
7. F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.
8. F. Cristian. Group, majority, and strict agreement in timed asynchronous distributed systems. In *Proceedings of the 26th Symposium on Fault-Tolerant Computing*, pages 178–187, Sendai, Japan, Jun 1996.
9. F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in air traffic control systems. *ACM Transactions on Computer Systems*, 14(3), Aug 1996.
10. F. Cristian and C. Fetzer. The timed asynchronous system model. Technical Report CSE97-519, Dept of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, 1997.
11. F. Cristian and S. Mishra. The pinwheel asynchronous atomic broadcast protocols. In *Proceedings of the Second International Symposium on Autonomous Decentralized Systems*, pages 215–221, Phoenix, AZ, Apr 1995.

12. F. Cristian, S. Mishra, and G. Alvarez. High-performance asynchronous atomic broadcast. *Distributed Systems Engineering*, 4(2):109–128, Jun 1997.
13. F. Cristian and F. Schmuck. Agreeing on processor-group membership in asynchronous distributed systems. Technical Report CSE95-428, Dept of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, 1995.
14. C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, Newport Beach, CA, 1995.
15. C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. In *Proceedings of the 15th ACM Symposium on Principles of Distributed Computing*, Philadelphia, PA, May 1996.
16. C. Fetzer and F. Cristian. Derivation of fail-aware membership service specifications. In *Proceedings of the 3nd Annual Workshop on Fault-Tolerant Parallel and Distributed Systems*, Orlando, Florida, Apr 1998. Available at http://www-cse.-ucsd.edu/users/cfetzer/DFAMS/dfams.html.
17. C. Fetzer, S. Mishra, and F. Cristian. The timewheel asynchronous group communication protocol. Technical Report CSE95-411, Dept of Computer Science and Engineering, University of California, San Diego, La Jolla, CA, 1995.
18. 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.
19. S. Mishra, C. Fetzer, and F. Cristian. The timewheel asynchronous atomic broadcast protocol. In *Proceedings of the 1997 International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1239–1248, Las Vegas, NV, Jun 1997.
20. S. Mishra, L. Peterson, and R. Schlichting. A membership protocol based on partial order. In *Proceedings of the Second Working Conference on Dependable Computing for Critical Applications*, pages 137–145, Tucson, AZ, Feb 1991.
21. S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering*, 1(2):87–103, Dec 1993.
22. S. Mishra and R. Yang. Thread-based vs event-based implementation of a group communication service. In *Proceedings of the 12th International Parallel Processing Symposium & 9th Symposium on Parallel and Distributed Processing*, Orlando, FL, Apr 1998.
23. L. Sabel and K. Marzullo. Election vs consensus in asynchronous systems. Technical Report TR95-1488, Cornell University, 1995.

This article was processed using the LaTeX macro package with LLNCS style