The Timewheel Group Communication System

Shivakant Mishra, Member, IEEE Computer Society, Christof Fetzer, Member, IEEE Computer Society, and Flaviu Cristian

Abstract—This paper describes a group communication system, called the timewheel group communication system, that has been designed for a timed asynchronous distributed system model. All protocols in the timewheel group communication system have been designed to be fail-aware in the sense that a process can detect, at any point in time, whether any of its properties is violated. Although these protocols have been designed to operate in an asynchronous distributed computing environment, they provide timeliness properties. The timewheel group communication system provides nine group communication semantics that a user can dynamically choose from while broadcasting an update. This system provides high throughput, fast delivery and stability times, uses a small number of messages per update broadcast, and evenly distributes the processing load among group members.

Index Terms—Group communication, timed asynchronous distributed system, high availability, fault tolerance, replication.

1 INTRODUCTION

A group communication service is a set of fault-tolerant protocols that enable replicated application processes to maintain a consistent replicated state despite random communication delays or failures. In this paper, we present the timewheel group communication system that is comprised of an atomic broadcast protocol and a clock synchronization protocol [22]. The timewheel group communication system provides four unique characteristics that distinguish it from other group communication services [12], [8], [37], [42], [9], [3], [34], [4], [20], [40], [38], [18], [5], [7]. First, this system has been designed for a timed asynchronous distributed system model [17]. This model allows the construction of dependable protocols that specify what outputs and state transitions should occur in response to inputs as well as a real-time interval in which these outputs and state transition are supposed to occur. Although the timewheel group communication system has been designed and implemented for a nonsynchronous distributed system, its properties are based on a timeliness parameter.

Second, the three protocols that comprise the timewheel group communication system are fail-aware [23]. A fail-aware service is one that can detect, in a timely fashion, when any of its properties is violated, e.g., due to excessive performance failures. Third, the timewheel group communication system supports multiple (nine) group communication semantics simultaneously. A user can dynamically choose any one of these nine semantics while broadcasting an update. This feature addresses the fundamental issue of what group communication semantics should be provided and how to accomplish a good overall performance for the supported semantics. On one hand, not all applications need strong semantics and, in this case, application designers would like to trade strong semantics for a better performance [13], [7]. On the other hand, when applications need stronger semantics, it is difficult and expensive to implement them on top of a reliable broadcast system with weak semantics. The nine group communication semantics provided by the timewheel group communication service allows application builders to choose the weakest possible group communication semantics that is sufficient for the correctness of their application.

Finally, one of these semantics preserves causal order between updates that may arise due to “hidden” channels [32]. Hidden communication channels involve communication between group members that may occur via mechanisms outside the group communication service. For example, two group members may communicate with one another using a network file system. To the best of our knowledge, this is the first group communication service that provides this semantics in an asynchronous distributed system.

In addition to these four unique characteristics, the timewheel group communication system provides excellent overall performance that includes fast delivery and stability times, a small number of message exchanges per update broadcast, even distribution of processing load among group members, and high throughput. This excellent overall performance is maintained in the absence of any failures and normal update arrival rates, in the presence of communication or process failures, and under very fast or very slow update arrival rates.

2 SYSTEM MODEL

The timed asynchronous system model [17] is the foundation of the timewheel group communication system. A timed asynchronous distributed system consists of a finite set of processes linked by an asynchronous datagram service. A one-way time-out delay $\delta$ is defined for the datagram service. Although there is no guarantee that a
message sent using this datagram service will be delivered within $\delta$ time units, it is likely to be delivered within $\delta$. We say that a process receives a message $m$ in a timely manner if the transmission delay of $m$ is not greater than $\delta$. When the transmission delay of $m$ is greater than $\delta$, we say that $m$ has suffered a performance failure or $m$ is late. We assume that the messages can also suffer omission failures [14] in addition to performance failures. This implies that a message sent by a process may never arrive at the intended receiver. However, whenever a message is received by the intended receiver, it is received uncorrupted.

The process management service defines a maximum scheduling delay $\sigma$, meaning that a process is likely to react to any trigger event (such as a timer event) within $\sigma$ time units. If a process $p$ takes more than $\sigma$ time units to react to a trigger event, it suffers a performance failure. When $p$'s reaction time is at most $\sigma$, we say that $p$ is timely. We assume that a process may suffer crash failures [14] in addition to performance failures. A process crashes by stopping to execute its program, e.g., a crashed process stops responding to any inputs. The model assumes that processes do not perform any incorrect state transitions.

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 known constant $\rho$. 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 $\rho$ is of the order of $10^{-4}$ to $10^{-6}$. We assume that hardware clocks have $\text{crash}$ failure semantics and that each noncrashed process has a correct hardware clock, i.e., a crash of a hardware clock $H_p$ results in the crash of $p$.

The timed model was carefully defined such that the probability that one of its assumptions is violated during runtime can be made negligible. The assumptions of the timed model can be seen as requirements that are implemented in software and hardware. Depending on the criticality of the applications, one can use different implementations of these requirements. For example, an important assumption of the timed model is that a process can use its hardware clock to measure time with a known maximum error. For safety critical applications, the can be implemented using redundant hardware clocks [25], while, for less critical applications, the use of a nonredundant hardware timer might be sufficient.

### 2.1 Synchronous versus Asynchronous Systems

Even though the timed model uses hardware clocks and has a notion of performance failures, it is not a synchronous system model. Defining the difference between a synchronous and an asynchronous system model is not straightforward. Because one can often use a software layer running on one model to implement the requirements of another model, the difference between a synchronous and an asynchronous system should not be based on specific assumptions of the model (e.g., if the model has a notion of time). Instead, it should be based on the power of a model (i.e., what problems can be solved in the model).

In our view, the difference between a synchronous and an asynchronous system model is that, in a synchronous system, one can guarantee that actions are executed by a given deadline. In an asynchronous system, one cannot guarantee that! More precisely, we call a system model a synchronous model if and only if 1) in any execution permitted by the model, there is always at least one correct process and 2) one can correctly implement the following task $T_S$: If and only if a correct process $p$ calls a function $f(q)$ at time $s$, then process $q$ executes a function $g(p)$ by time $t$ such that $s < t < s + \Delta$ where $\Delta$ is a constant. In a partially synchronous system, an implementation of $T_S$ is eventually correct, i.e., if $s \geq u$, where $u$ depends on the execution. We call a system model that is neither synchronous nor partially synchronous an asynchronous model. The timed model is asynchronous because it neither excludes complete system failures, i.e., it permits executions in which all processes are crashed nor does it permit to implement task $T_S$.

### 2.2 Circumventing Impossibility Results

It has been shown that one cannot solve consensus [28] or group membership [10] in asynchronous systems. This is due to the liveness conditions used in the problem specifications: “eventually something good (e.g., installation of a new membership) has to be achieved.” In the timed model, we circumvent these impossibility results by using conditional timeliness requirements of the form “if the system behaves well in a given time interval $I$ of length $l$, then something good has to be achieved within $I$.” Since a protocol only has to achieve something good if the system behaves well (e.g., like a synchronous system) sufficiently long, one can actually solve a large set of practically relevant problems.

To permit the definition of when a system behaves well, the timed model has the notion of message and process performance failures. For example, in this paper, we say that the system behaves well in a given interval $I$ if a majority $M$ of the processes do not suffer performance failures in $I$ and they can communicate with each other in a timely fashion during $I$. Furthermore, all messages that arrive from processes that are not in $M$ are detectably late, i.e., the processes in $M$ will drop these messages on reception. The idea of this definition is that, while some processes are partitioned away from the majority, e.g., due to a network failure, the system must still make progress.

### 2.3 Fail-Awareness

Our approach to designing protocols for the timed model typically uses a step-by-step refinement of the problem specification (e.g., see the stepwise refinement of a group membership specification in [24]). We first specify the protocol for a completely synchronous system (i.e., no crash, omission, or performance failures) and then, step-by-step, modify the specification such that it becomes implementable in a timed asynchronous system. To circumvent impossibility results while still ensuring the safety of the system, we introduced the notion of fail-awareness [27].

Most applications require the enforcement of safety properties, i.e., properties that must always hold. For example, a safety property could state that “whenever a train is in the railway crossing, the gates of the crossing are down.” Implementing such safety properties often requires that the underlying system be synchronous, e.g., because sensor data has to be communicated and processed in a timely fashion. However, adding a simple “indicator” to such a safety
property makes the new property implementable in timed asynchronous systems. If the indicator is true, the property is guaranteed to hold. Whenever the underlying system behaves well sufficiently long, the indicator must be true.

The fail-aware property of the previous railway crossing example becomes: “If the indicator I is true, then, whenever a train is in the railway crossing, the gates of the crossing are down.” The intuition is that as long as the indicator is true, the safety property is guaranteed to be true. However, if the indicator becomes false, the safety property might be violated. In this case, the systems might have to be switched to a fail-safe state, e.g., by lowering the gates. This switching to a fail-safe state can be enforced by using an appropriate interface to the environment (see [27]). Using fail-awareness not only makes sense for fail-safe, but also for fail-operational systems. It can be used to make fail-operational systems robust against unexpected performance and omission failures.

2.4 Parameter Selection

The timed model has three parameters: $\delta$, $\sigma$, $\rho$. Parameters $\delta$ and $\sigma$ are used to define message and process performance failures. The actual values of these parameters must not affect the correctness of an application! Larger values will make a system “behave well” more often, while smaller values will make the system “behave well” less often. Due to the use of conditional timeliness requirements and fail-aware safety properties, we avoid having the correctness of an application depend upon the values of $\delta$ and $\sigma$.

The selection of good values for $\delta$ and $\sigma$ is nevertheless important to increase the liveness of a system, i.e., the periods in which all indicators are true. For real-time applications, one can often derive values for $\delta$ and $\sigma$ from their specification and the used protocol. For example, if the delay of a protocol is $5\delta + 3\sigma$ and the delay should be at most 5 seconds, one can derive the constraint $5\delta + 3\sigma \leq 5s$ (see also [17]). Multiple such constraints usually permit the derivation of appropriate bounds for $\delta$ and $\sigma$.

Alternatively, an application designer could define a “target” probability for performance failures, i.e., associate term “likely” (which we use in the model description) with a probability. Measurements of the transmission and scheduling delays in the target system together with the given target probability permits one to define $\delta$ and $\sigma$. However, we do not believe that the probability of failures should be part of the model: 1) To make applications more robust, we do want to avoid having the correctness of an applications depend on the probability of failures (e.g., we want to disallow simple protocols that “guarantee” certain communication by simply transmitting messages $F + 1$ times) and 2) force a protocol to adapt to changing environments.

The correctness of fail-aware protocols typically depends on a correctly chosen value for the clock drift $\rho$. The clock drift $\rho$ can typically be derived from the physical properties (or the data sheet) of the used oscillators and the operating temperature range of the target system. Due to reliance on the bounded drift rate of clocks, for critical applications, one might use a fault-tolerant hardware clock design [25].

2.5 Related Models

2.5.1 Time-Free Model (a.k.a. FLP Model)

The timed asynchronous distributed system model is significantly different from the well-known time-free asynchronous distributed system model [28], 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 dependable distributed protocols that are of importance in practice, such as consensus, election, or membership, are not implementable [28], [39], [10] in this system model. Most existing distributed systems based on non-real-time operating systems and communication services, such as Unix and UDP, can be viewed as timed asynchronous.

2.5.2 Timely Computing Base

The timely computing base (TCB) [41] splits a system into two parts: a synchronous control part and an asynchronous payload part. We agree with the authors that it is useful and sometimes even necessary to divide a system into synchronous and asynchronous parts. However, we suggest partitioning the system in a different way.

We propose dividing a critical system into fail-safe and fail-operational components [27] instead of dividing it into a control and a payload part. Each of these components might be further dividable into fail-operational and fail-safe components. One can continue this division until the system consists of atomic fail-operational and fail-safe components, i.e., it is not reasonable to divide these components any further.

We suggest designing both fail-operational and fail-safe components using the fail-aware design paradigm. The atomic fail-operational components should be executed in parts of the underlying computing systems that are made as synchronous as possible/needed. For example, one might use reserved processors and network links to execute the fail-operational components. We propose using the fail-aware design paradigm also for the fail-operational components because this permits us to detect when the underlying system is not sufficiently synchronous and, in turn, permits us to reconfigure the system to reduce the occurrence of further performance and omission failures. Since components typically interact with each other, the indicators of a fail-aware service permit a client to detect that the service cannot guarantee its safety properties and, in this way, a client can avoid state contaminations.

To increase the robustness of the system, we do not agree that the inclusion of assumptions Ps1 (timely processing) and Ps3 (timely message delivery) in the TCB [41] is a good idea. While we agree that Ps1 and Ps2 are requirements that at least parts of the underlying system should attempt to implement (e.g., those parts on which fail-operational components are running), a protocol should not depend on these assumptions since this can decrease the robustness of the protocol and can be the cause of state contamination problems. The authors of [41] argue that Ps1 and Ps2 are needed to guarantee the semantics of fail-operational

---

1. Reference [17] defines a fourth parameter, $\delta_{\min}$ (the minimum message transmission delay), which can safely be set to 0.
components. While this is superficially true, the real issue is how to handle the case if Ps1 and Ps2 are violated during runtime. The fail-awareness approach will also guarantee that all safety and timeliness properties hold as long as Ps1 and Ps2 are true. However, if Ps1 and Ps2 are violated during runtime, the system can detect this (using the indicators of the fail-aware services) and can then attempt to repair the problem (e.g., by shedding load). After a successful repair, the fail-aware services are forced to provide their properties again within a bounded time.

Adding services to a model is important to simplify the design and implementation of protocols. Instead of proposing a fixed set of services that are part of the TCB, we proposed different fail-aware services for the timed model that provide similar functionalities (e.g., [26], [24]). This permits a system designer greater flexibility in selecting the right set of services for the problem at hand. Also, this paper describes group communication services that can be used to simplify the design of protocols for the timed model.

2.5.3 Synchronous Model
An alternative approach to designing real-time applications is proposed by [30]. The authors propose designing protocols for a time-free model extended by a crash failure detector and then immersing this algorithm into a synchronous model during the implementation phase. The authors claim that, during this immersion, the timeliness properties of the protocol will emerge.

The approach proposed by [30] could be viewed as the inverse of our approach. They propose using a weaker design model and then running the protocol in a synchronous system model. The hope is that, since the protocol was designed for a time-free model, it can naturally cope with performance failures that might occur during runtime. We proposed [24] starting by specifying the problem for a completely synchronous system and then, step-by-step, refining the design/specification while weakening the system model. Our approach is a good match with evolutionary software processes (e.g., extreme programming) that are used in practice. One can learn how to solve a simpler problem and use the acquired knowledge to solve a harder problem in the next step.

Finally, by restricting the design to a time-free model, one cannot use hardware clocks and, hence, clock-based mechanisms like a lease [29]. Leases are a very elegant and useful tool and, in the design of our protocols, we make intensive use of leases. Leases can often be used to replace crash failure detectors and timing failure detectors. Using crash and timing failure detectors to replace leases is often impossible or at least very awkward.

Another difference between our approach and that of [30] is the use of a feasibility analysis to determine the parameters of the system, e.g., δ. While we do not suggest the use of a feasibility analysis, a feasibility analysis can also be combined with our approach if needed. However, a feasibility analysis might not always be possible since the feasibility conditions of COTS components are often not known. The authors of [30] acknowledge that feasibility conditions cannot guarantee the timeliness (and, hence, the safety) of the system. For most systems, it seems, therefore, to be more cost effective to use a measurement-based approach (possibly combined with fault-injection) to determine the parameters of the model instead of a feasibility analysis since the safety of the system has to be guaranteed in some other way anyhow. The measurements can be used to select appropriate model parameters to make sure that the system is sufficiently live.

3 SEMANTICS AND PERFORMANCE
3.1 Semantics
The group communication semantics that we consider belong to three classes: termination, atomicity, and order. The Timewheel group communication service provides three atomicity semantics—weak atomicity, strong atomicity,
and strict atomicity—and three order semantics—unordered, total order, and time order. The combination of different semantics from the three classes results in nine group communication semantics.

### 3.1.1 Termination

The termination semantics specifies which updates have to be delivered by processes. It requires that all updates proposed by $\Delta$-stable processes are eventually delivered by all $\Delta$-stable processes. The predicate $\Delta$-stable is defined in [22]. A process $p$ is said to be $\Delta$-stable iff 1) $p$ is timely, 2) $p$ can communicate in a timely manner with a majority of processes in the team and all these processes are $\Delta$-stable, and 3) $p$ can detect all messages from non-$\Delta$-stable processes as being late and can reject them.

### 3.1.2 Order

The order semantics determines the order in which different updates are delivered. The unordered delivery allows group members to deliver updates in any order. In particular, different group members may deliver updates in different orders. Total order delivery ensures that 1) all group members deliver all updates proposed with total order semantics in the same order and 2) that this order preserves the FIFO property, i.e., when a process $p$ proposes updates $u_1$ followed by $u_2$ and a process $q$ delivers both of these updates, then $q$ must deliver $u_1$ before $u_2$. Time order delivery ensures that updates are delivered in the order of their send timestamps, where the send timestamp of an update $u$ is the value displayed by the local clock (which is synchronized by the fail-aware clock synchronization service) at the time when the broadcast of $u$ is initiated. Time ordered delivery guarantees total order between updates delivered to different teams.

In defining total and time orders, causal order [8] between updates being broadcast plays an important role. Causal order is defined using the “happened before” relation [32]. An event $A$ happened before another event $C$, iff 1) $A$ and $C$ are events in the same process and $A$ occurs before $C$ or 2) $A$ denotes the event when a process communicates some information $I$ and $C$ denotes the event when a process gets information $I$ or 3) there exists an event $B$ such that $A$ happened before $B$ and $B$ happened before $C$. Causal order has been proposed as a mechanism to build dependable systems [36], [8], [9], [34]. It ensures that if the proposal of an update $u_1$ happened before the proposal of an update $u_2$ and $u_2$ is delivered by a process $q$, then $q$ must deliver $u_1$ before $u_2$.

Processes may communicate using services other than the group communication service, e.g., they could use a network file system. These other communication services are called hidden communication channels. Causal delivery of updates requires processes to ensure the following delivery order: Consider the following scenario: A process $p$ proposes an update $u_1$ and then communicates information $I$ to another process $q$ using a hidden communication channel. After receiving $I$, $q$ proposes an update $u_2$. Causal delivery of updates in the presence of hidden communication channels requires that any process that delivers $u_2$ must deliver it only after delivering $u_1$. Synchronized clocks can be used to guarantee causal delivery in the presence of hidden communication channels. A requirement for this is that the minimum communication delays of hidden communication channels must be greater than the maximum deviation between clocks [32]. The Timewheel group communication service does not explicitly provide a separate causal order semantics. However, the total and time order can be combined with an appropriate atomicity semantics to ensure causal delivery.

### 3.1.3 Atomicity

The atomicity semantics requires processes to agree on the updates they deliver. Weak atomicity requires that, when a group member $p$ delivers an update $u$, then either all group members deliver $u$ or a new group $q$ is created that does not contain $p$ as a member and no member of $q$ delivers $u$. 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 remaining group members form a new group and deliver $u_2$, but not $u_1$. This may lead to an inconsistent system state because group members have delivered an update $u_2$ without delivering an update $u_1$ on which $u_2$ potentially depends. In particular, weak atomicity is not sufficient to guarantee causal delivery.

Strong atomicity satisfies the weak atomicity semantics. In addition, it requires that if a process $p$ first delivers an update $u_1$ and subsequently proposes an update $u_2$, then any process that delivers $u_2$ must deliver $u_1$ first. To illustrate this property, let us assume that no recoveries occur and all updates are delivered in total order. There exists a sequence $H$ consisting of updates and membership changes such that a majority of processes deliver these updates and membership changes in the order given by $H$. In addition, 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 delivering $u_1$. This means that strong atomicity cannot guarantee causal delivery in the presence of hidden channels.

Strict atomicity satisfies strong atomicity. In addition, it requires that a process $p$ deliver an update $u$ only when a majority of processes deliver $u$. The inconsistent behaviors described for weak atomicity and strong atomicity cannot happen in strict atomicity. This is because those inconsistent behaviors were possible due to the fact that a single group member (or a minority subset of group members) could deliver an update without knowing if other group members have delivered that update. With strict atomicity, a majority of processes has to deliver any update and, because a group must contain a majority of the processes, all group members have to deliver that update. Since the Timewheel group communication service assumes synchronized clocks, time
ordered delivery, together with strict atomicity, guarantees causal delivery, even in the presence of communication through hidden communication channels.

3.2 Performance Metrics
We consider the following important performance indices: average broadcast delivery time, average broadcast stability time, average number of messages needed to broadcast an update, throughput, and the distribution of processing load among group members. The broadcast delivery time of an update is defined to be the duration between the moment that update is entrusted to the broadcast service by a group member and the moment that 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 broadcasts an update in a local buffer until they learn that the update is stable, that is, it was received by all group members. The broadcast stability time of an update is the duration between the moment a broadcast server receives that update to be broadcast and the moment all broadcast servers learn that the update is stable. 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 throughput is defined as the number of updates delivered per second by each group member for a given update arrival rate. The distribution of processing load among group members refers to how busy a member is in relation to other members, over some sufficiently long period of time (so that all group members initiate about the same number of broadcasts during that period).

4 INFORMAL OVERVIEW

The timewheel group communication system consists of a clock synchronization protocol, an atomic broadcast protocol, and a group membership protocol. It depends on an unreliable datagram protocol such as UDP for communication. Fig. 1 shows the dependency relation between these protocols.

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 $\Psi$ and 2) the drift rate of the synchronized clocks is within a known linear envelope of real time. In the timed asynchronous distributed system model, it is not possible to keep correct clocks synchronized all the time. This is because this model allows a (very unlikely) run in which none of the processes can communicate with any one another. We therefore use a fail-aware clock synchronization protocol [22] 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, described in detail in Section 5, is an optimization and extension of the protocols proposed in [12] and [18]. 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 1) associate unique numbers, called ordinals, to updates/membership changes, 2) establish the stability of broadcast updates, and 3) detect message losses. A group member, called the decider, is responsible for sending decision messages. A decision message includes an ordering and acknowledgment list (referred to as oal henceforth) consisting of update/membership change descriptors, along with information 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. This decision message also is used to maintain the liveliness of the group communication service during idle periods. 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, are satisfied.

The timewheel group membership protocol, described in detail in Section 6, is a majority agreement protocol [15]. It 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 the predicate $\Delta$-stable [22] defined in Section 3. 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.

2. The delivery order of updates/membership changes is not necessarily the same as the order of the ordinals associated with them.
The group membership problem or the atomic broadcast problem is not solvable in a time-free asynchronous system model [28], [11]. However, existing asynchronous systems typically have 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 $\Delta$-stable interleaved by relatively short periods in which the system is not $\Delta$-stable. We formalize this observation by a progress assumption [21]: We assume that the system will be infinitely often $\Delta$-stable. This progress assumption allows a deterministic solution of consensus [21]. Since the consensus problem is as hard as the group membership or the atomic broadcast problem [11], it also allows a deterministic solution of the group membership or the atomic broadcast problem.

## 5 Timewheel Atomic Broadcast

To broadcast an update $u$ at local time $ts$ with order $order$ and atomicity $ato$, a member $s$ in group $g$ disseminates a proposal message $<g,u,s,n,ts,hdo,hero,order,ato>$ to all other group members; $ts$ is called the send timestamp of $u$. Each group member in $g$ is assigned a unique rank $s$, $N > s \geq 0$, where $N$ is the total number of group members in $g$. 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 the decider, associates a unique natural number, called the ordinal, with each update/membership change; the ordering of these ordinals is independent of the atomicity and order semantics required by the client. The first update/membership change 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, $hero$, in a proposal message specifies that the proposer has received all proposals/membership changes with ordinals smaller than or equal to $hero$; it is used to minimize stability times.

A positive acknowledgment scheme is used to recover from communication failures and establish the stability of updates. A decider assigns ordinals 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 $hero$ greater than or equal to $o$. To bound the stability time, a process must acknowledge all received proposals periodically. To facilitate this and to evenly distribute the communication overhead of sending decision messages among group members, the role of the decider is rotated between group members along a logical ring that defines a predecessor/successor relation among group members. A 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 member can be a decider is bounded by an a priori given constant $D$ (called the decider timeout period).

A decision message is a tuple $<ord, oal>$, where the ordering and acknowledgment 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 $<s,n,ts,ack>$ identifies an update with id $<s,n>$ sent at time $ts$ by group member $s$. The acknowledgment 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 descriptor $<g,ack>$ identifies a group $g$ created by the membership protocol.

In the following, we use $oal$ to refer to the $oal$ of the current decider or the $oal$ included in the last decision message when there is no current decider. A decider modifies $oal$ by deleting stable proposals/membership changes from the head and appending new descriptors at the tail of this list. $oal$ and $ord$ implicitly associate an ordinal with each descriptor in $oal$; ordinal $ord + k$ with the $k$th descriptor in $oal$. 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 $<g,u,s,n,ts,hdo,hero,order,ato>$, $p$ 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 each of these descriptors, the send timestamp and the ordinal are 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 $mhero$ that records, for each member $r$, the maximum $hero$ 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) It sets $ts$ 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 decider role.

When a process \( d \) becomes the decider by receiving a decision \(< ord, oal, \rangle \), it first uses \( oal \) 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 \( mchcr[r] \).

An update proposed by a group member should be admitted to be delivered only if the sender has been a member of all groups formed since it proposed that update. For example, let process \( q \) be a member of groups \( g_1 \) and \( g_2 \), but not a member of group \( g_3 \), 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 two “deletion rules” which delete such proposals from the \( pb \) 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 \( oal \), 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 \( \geq 0 \) 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.

5.1 Delivery of Updates

A process \( p \) delivers an update \( u \) proposed by process \( s \) as soon as three deliver conditions—general, atomicity, and order—are satisfied for \( u \). The general delivery condition ensures that \( p \) does not deliver an update \( u \) which is later removed from the \( pdb \) by the deletion rules described above or which is later removed from the \( oal \) by the membership protocol. To do this, \( p \) must not deliver \( u \) when 1) \( u \) is erasable by the deletion rules or 2) \( u \) is marked as undeliverable by the membership protocol, 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 broadcast with a highest delivered ordinal \( hdo \) 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 \( pr \). In the absence of hidden communication channels, this condition ensures that process \( p \) has received all updates on which \( u \) could causally depend.

Each member maintains an integer \( hdo \) (highest delivered ordinal), which is included in every proposal message sent. When a member \( p \) delivers \( u \), it updates \( hdo \) by setting \( hdo \) to the maximum of \( hdo \) 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 than all updates it may depend on.

The atomicity condition for an update \( u \) that was broadcast 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 \( u \) by checking the \( ack \) vector of the proposal descriptor of \( u \) in \( pdb \). Finally, the atomicity condition for updates sent with weak atomicity is always true.

The order condition for updates with unordered delivery semantics is always true. The order condition for an update \( u \) with total order delivery is satisfied at \( p \) when \( p \) has delivered all updates with unordered/total order semantics that have ordinals smaller than that 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 \( s \) as 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 or receiving a decision message that contains the proposal descriptor of that proposal. Since a 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 \( u \) with time order delivery and with send timestamp \( ts \) is true iff 1) \( (ts, s) \leq \min\{\{hsts[r], r : \forall r\} \), 2) all updates with smaller send timestamps known to \( s \) have been delivered, and 3) 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 \( ts \).

5.2 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 \( hcdr \). Stability is used in purging proposal descriptors (maximum threshold), 2) the difference between the highest ordinal in the latest decision message and the \( hcdr \) of all updates with smaller send timestamps known to \( s \) have been delivered, and 3) if the maximum time until \( p \) will be the the decoder is greater than \( L \). time units. This ensures that silent group members send their \( hcdr \) 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 [35]. The purpose of flow control is to handle such situations. The flow control therefore piggy-backs on every decision message of the highest continuously delivered ordinal, \( hcdr \), i.e., all proposals with ordinals not greater than \( hcdr \) have been delivered by the decoder. 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 \( modu \). Therefore, a buffer overflow cannot occur at a group member if its proposal buffer can store at least \( N \times modu \) proposals and the proposal descriptor buffer can store \( N \times modu \) proposal descriptor-ordinal pairs.

6 Timewheel Group Membership

The timewheel membership protocol is a majority agreement protocol [15]: 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 [15]. More precisely, the membership protocol satisfies the following properties (see [24]):

1. If a process is \( \Delta \)-stable for at least \( \tau \) 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 \( \Delta \)-stable for at least \( \tau \) time units, it is included in any up-to-date group,
4. If a process’ current group has been out-of-date for \( \tau \) 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.

6.1 Overview

This protocol is based on the ideas developed in the membership protocols described in [3], [34], [4]. Informally, the key idea in these protocols is that the team members exchange membership messages and each live team member \( p \) maintains a set \( S_p \) of live team members based on these messages. The team members continue to exchange membership messages until the following property is satisfied: \( \forall q \in S_p, S_q = S_p \).

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 at taking 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.

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 a 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 that $q$ 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 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 $(N - 1)D$ 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 participates in a new election. Since $p$ will no longer assume the role of decider in the group resulting 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.

### 6.2 Detailed Description

#### 6.2.1 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 $\delta$ 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 receive 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 reports 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 $ts$ from the expected sender $s$, we implicitly assume that $p$
checks the send timestamp of \( m \) to determine that it hasn’t already received \( m \) and that \( m \) is sent after time \( ts \).

### 6.2.2 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 Fig. 2): join, failure-free, wrong-suspicion, 1-failure-receive, 1-failure-send, and \( n \)-failure.

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 a 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 wrong-suspicion states are used to handle a single omission/performance failure of a decision message (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 failure 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, \( \text{join-list}_p \), of all those processes from which \( p \) has received at least one join message in the last \( N - 1 \) slots; \( \text{join-list}_p \) 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. \( \text{join-list}_q \) contains a majority of the processes and
2. \( q \) has received a join message from each process \( p \) in \( \text{join-list}_q \) in \( p \)'s last time slot such that \( \text{join-list}_q = \text{join-list}_p \).

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 decider 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 \( q \) (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 \( q \) 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

---

*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.*
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 a 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.

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 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 last in at most \((N - 1)D\) time
units. During this waiting period, \(p\) transmits a reconfigura-
tion 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.

### 6.3 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 delivers an update whose proposal descriptor is
removed from oal.

#### 6.3.1 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 undeliver-
able 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 in which 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 \(pr\).

#### 6.3.2 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 reconfigura-
tion 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 ensure the atomicity con-
straints. View \(v_p\), together with field \(dpd\), contains
acknowledgments 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 $pb$.

7 PERFORMANCE

The timewheel group communication system has been implemented in three different forms. The first implementation runs on a network of SGI workstations (Indys and O2s) connected by a moderately loaded 10Mb/s Ethernet. The second implementation runs on a network of Windows/NT workstations. The third implementation uses CORBA [1] and runs on a network of SGI (Indys and O2s), Windows/NT, and Sun workstations. We report the performance measured from the first implementation. This implementation consists of about 2,000 lines of C++ code and uses the event-based implementation mechanism to handle concurrent events. We have reported absolute performance in this section. A comparative performance with other group communication services will be perhaps more useful, but we could not find an implementation of other group communication service on a similar computing platform.

7.1 Atomic Broadcast

We measured four performance indices: throughput, average delivery time, average stability time, and average number of messages needed per update broadcast. Because the timewheel group communication system provides several different group communication semantics, we describe its performance for different group communication semantics separately.

Figs. 3, 4, and 5 show the throughput for weak, strong, and strict atomicity semantics, respectively. The group size in all these measurements was three. For all semantics, the number of updates delivered per second is approximately equal to the total number of requests generated per second for lower arrival rates (e.g., up to about 500 requests generated per second for weak atomicity). Note that the flow control mechanism allows the timewheel system to refuse new proposals to avoid buffer overflows. As the request generation rate increases, the effect of processing and queuing delays causes the number of updates delivered per second to be smaller than the number of requests generated per second. For very high update arrival rates, the UDP message loss rate becomes significant due to the saturation of the Ethernet driver and is the dominant reason why throughput stops increasing.

Comparing the three figures, it is clear that the throughput decreases as the semantics range from no order to time order and from weak atomicity to strict atomicity. The throughput is highest for weak atomicity and no order semantics and lowest for strict atomicity and time order.
semantics. This is an expected result as the weak atomicity and no order are the weakest atomicity and order semantics, respectively, and the strict atomicity and time order are the strongest semantics, respectively.

Table 1 shows the average delivery times measured for (weak atomicity, total order), (strict atomicity, total order), (weak atomicity, no order), and (strict atomicity, total order) semantics. For all semantics, the delivery time is higher at low update interarrival times. It levels off with an increase in the update interarrival time. The reason for large delivery times at low update interarrival times is that an update may arrive at a group member before that member has finished processing one or more updates which arrived earlier. As a result, an update may have to wait for some time at a group member for processing at low update interarrival times. Also, similarly to the observation in throughput measurement, the average delivery time increases as the group communication semantics vary from no order to time order and from weak atomicity to strict atomicity.

Table 2 shows the average stability times measured for (weak atomicity, no order), (weak atomicity, total order), (strong atomicity, time order), and (strict atomicity, no order) semantics. Again, we notice that the stability time is higher at low update interarrival times and it levels off with an increase in the interarrival time. A similar pattern was observed in the average delivery time and the reason for this is the same as that given for the average delivery time.

Finally, the average number of messages exchanged per atomic broadcast is shown in Table 3 for (weak atomicity, no order) semantics. There was very little variation in the average number of messages exchanged per atomic broadcast between different group communication semantics. The average number of messages exchanged per atomic broadcast increases with an increase in the interval arrival time. The reason for this pattern is that as the interval arrival time increases, there are fewer updates broadcast. As a result, the decision message rotates with fewer and fewer updates with increase in the interval arrival time. This results in an increase in the average number of messages exchanged per atomic broadcast.

### Table 1: Delivery Time (msec)

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Interarrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>Order</td>
</tr>
<tr>
<td>Weak</td>
<td>Total</td>
</tr>
<tr>
<td>Strong</td>
<td>Time</td>
</tr>
<tr>
<td>Strict</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Total</td>
</tr>
</tbody>
</table>

### Table 2: Stability Time (msec)

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Interarrival Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>Order</td>
</tr>
<tr>
<td>Weak</td>
<td>Total</td>
</tr>
<tr>
<td>Weak</td>
<td>Total</td>
</tr>
<tr>
<td>Strong</td>
<td>Time</td>
</tr>
<tr>
<td>Strict</td>
<td>No</td>
</tr>
</tbody>
</table>

### Table 3: Number of Messages for (Weak Atomicity, No Order) Semantics

<table>
<thead>
<tr>
<th>Interarrival Time</th>
<th>Number of Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.0</td>
<td>2.0</td>
</tr>
<tr>
<td>5.0</td>
<td>2.9</td>
</tr>
<tr>
<td>8.0</td>
<td>4.0</td>
</tr>
<tr>
<td>10.0</td>
<td>5.0</td>
</tr>
</tbody>
</table>

7.2 Group Membership

We have measured the effect of the membership protocol on service performance during the failure-free period and the time it takes for a new group to be formed after a member failure. We noticed that the group membership protocol did not affect throughput, delivery time, stability time, or number of messages exchanged in any significant way during failure-free periods.

The time to create a group after a member failure depends on three factors: team size, decider timeout period ($D$), and number of simultaneous failures. Table 4 shows the average time it takes to create a group after a single failure, two-process failure, and three-process failure for various values of team size and decider timeout period. To simulate the failure of a process, we simply killed that process. The time to create a group after a failure depends on precisely when a process is killed during a time slot. We had no way of controlling this precise time in our experiments. As a result, the time to create a group that we measured varied significantly (by 2-3 milliseconds) over different runs. The average time reported in Table 4 was calculated after running each experiment 20 times.

There are three observations we make from these measurements. First, there is a significant difference in the time to create a group between a single process failure case and a multiple process failure case. The reason for this is that the membership protocol has been optimized for a single failure case. The main delay in this case is the time it takes for
the first process to detect the failure. This time is set to twice the decider timeout delay (2D). As a result, the time to create a new group in this case is little more than 2D.

In case of multiple simultaneous failures, the n-failure protocol is initiated only after a process detects a second failure. This may take up to four times the decision timeout period. As a result, the time to create a new group is a little more than 4D. It is important to note that the difference in time to create a new group between two simultaneous failures and more than two simultaneous failures is not expected to be much. This is evident from the three simultaneous failure experiments that we performed. The reason for this is that the n-failure protocol is initiated as soon as a second failure is detected, irrespective of the total number of simultaneous failures.

Second, the decision timeout period is the dominant factor in determining the time to create a new group after one or more failures. The other factor that affects this time is the communication delay between processes. Typically, communication delay is an order of magnitude smaller than the decision timeout period. Finally, an increase in team size increases (slightly) the time to create a group. The reason for this is that, with an increase in team size, more processes need to agree on the new group membership. This requires more message exchanges and results in an increased time to create a new group.

8 Conclusion

Group communication services have proven to be extremely useful for constructing highly available and dependable distributed applications. Because of their usefulness, a large number of group communication services have been proposed, designed, and implemented. Notable ones include Isis [8], [9], Delta-4 [37], Transis [3], Consul [34], Totem [4], Horus [40], Pinwheel [18], and Spread [5], [2]. The major limitation of these services is that they do not provide any timeliness guarantees and, hence, cannot be used for constructing real-time, dependable distributed applications. On the other hand, group communication services that provide timeliness guarantees, such as Mars [31] and AAS [6], [16], are designed for a synchronous distributed system. The key limitation of these services is that they cannot be used on commonly available distributed computing environments, but they cannot be used to construct real-time, dependable distributed applications. On the other hand, we have group communication services that can be used to construct real-time, dependable distributed applications, but they cannot operate on an asynchronous distributed computing environment.

The timewheel group communication service proposed in this paper addresses this limitation of group communication services. It operates in a (timed) asynchronous distributed computing environment and provides timeliness guarantees. The key characteristic of this service is that the protocols that comprise this service are fail-aware. This means that a process can determine if it is satisfying the specified timeliness properties at any time. In case it determines that it cannot satisfy those properties, perhaps due to excessive performance failures, it can inform the application about this situation. This fail-awareness property [23] is made possible by the fact that the commonly available distributed computing environments have been modeled as timed asynchronous distributed computing environments. These computing environments were traditionally modeled as asynchronous distributed computing environments that had no timeliness specifications. The timed asynchronous distributed computing model [17] recognizes that fact that, although, in most distributed computing systems, communication delays or process scheduling delays are unbounded, they are likely to be less than some time bound most of the time. This observation allows us to build fail-aware applications with timeliness guarantees.

In addition to the timeliness and fail-awareness properties, the timewheel group communication service provides support for nine group communication semantics simultaneously, preserves causal order between updates that may arise due tohidden communication channels, and provides excellent overall performance that includes fast delivery and stability times, a small number of message exchanges per update broadcast, even distribution of processing load among group members, and high throughput. At present, we have used this service to construct two dependable distributed applications: an availability management service [33] and a stable storage service [19]. Both of these applications vary in their semantic requirements and demonstrate the usefulness of multiple group communication semantics.

Acknowledgments

This work was supported in part by US Air Force Office of Scientific Research grants F49620-96-1-0103 and F49620-98-1-0070 and US National Science Foundation grant CDA-9617197.

References

MISHRA ET AL.: THE TIMEWHEEL GROUP COMMUNICATION SYSTEM


---

Shivankant Mishra received the BTech degree in computer science and engineering from the Indian Institute of Technology, Bombay, in 1985, the MS degree in computer science from Southern Illinois University at Carbondale in 1987, and the PhD degree in computer science from the University of Arizona, Tucson, in 1992. Currently, he is an associate professor in the Department of Computer Science, University of Colorado, Boulder. His research interests include design, implementation, and evaluation of dependable distributed systems, communication protocols, and operating systems. The current focus of his research is in the design and implementation of system-level support to facilitate dependable, high performance computing over the Internet. He is a member of the IEEE Computer Society.

Christof Fetzer received the PhD degree from the University of California, San Diego, in 1997. He is a senior technical staff member in the dependable distributed systems group at AT&T Labs-Research. He has published more than 35 papers in the field of dependable systems. He won two best student paper awards and a Marther Premium from IEEE. He has been a program committee member for various conferences and workshops. His current research interests are pervasive dependability and fail-operational system design. He is a member of the IEEE Computer Society.

Flaviu Cristian passed away in April 1999. At the time of his death he was a professor in the Department of Computer Science and Engineering, University of California, San Diego.