Synchronous and Asynchronous Group Communication (Long Version)

Flaviu Cristian
Deptm of Computer Science and Engineering
University of California, San Diego
flaviu@cs.ucsd.edu

In distributed systems, high service availability can be achieved by letting a group of servers replicate the service state; if some servers fail, the surviving ones know the service state and can continue to provide the service. Group communication services, such as membership and atomic broadcast, have been proposed to solve the problem of maintaining server state replica consistency. Group membership achieves agreement on the history of server groups that provide the service over time, while atomic broadcast achieves agreement on the history of state updates performed in each group.

Since many highly available systems must support both hard real-time and soft real-time services, it is of interest to understand how synchronous (hard real-time) and asynchronous (soft real-time) group communication services can be integrated. We contribute towards this goal by proposing a common framework for describing properties of synchronous and asynchronous group communication services and by comparing the properties that such services can provide to simplify the task of replicated programming.

1 Introduction

In distributed systems, high service availability can be achieved by replicating the service state on multiple server processes. If a server fails, the surviving ones continue to provide the service because they know its current state. Group communication services, such as membership and atomic broadcast, simplify the maintenance of state replica consistency despite random communication delays, failures and recoveries. Membership achieves agreement on the server groups that provide the service over time, while atomic broadcast achieves agreement on the history of state updates performed in these groups.

Since many highly available systems must provide both hard and soft real-time application services, it is of interest to understand how synchronous (hard real-time) and asynchronous

Invited paper IEEE Workshop on Fault-tolerant and Parallel Distributed Systems, April 15, Honolulu, Hawaii. This work was partially supported by IBM and by grants from the Air Force Office of Scientific Research and Sun Microsystems.

(soft real-time) group communication services can be integrated. This paper attempts to contribute towards this goal, by proposing a common framework for describing properties of synchronous and asynchronous group communication services and comparing the properties that synchronous and asynchronous group communication can provide to simplify replicated programming. The paper reflects our practical experience with the design of synchronous and asynchronous group communication services for a complex system for air traffic control, the Advanced Automation System[1] [13].

For simplicity, we consider a unique application service S implemented by servers replicated on a fixed set of processors P. The servers (one per processor) form the *team* of S-servers. The one-to-one correspondence between servers and processors allows us to ignore the distinction between server groups and processor groups and the issues related to multiplexing server level broadcasts and groups on top of processor level broadcasts and groups.

2 Asynchronous System Model

Team processors do not share storage. They communicate only by exchanging messages and by measuring the passage of time. Processors exchange messages via a *datagram* communication service. Messages can get lost and communication delays are unbounded, although *most*[2] messages arrive at their destination within a known *timeout* delay constant d [6]. Thus, datagram communication has *omission/performance* failure semantics [10].

Processors have access to *stable storage* and *hardware clocks*. Clocks measure time with a known accuracy by running within a linear envelope of real-time. Servers are scheduled to run on processors in response to trigger events such as message arrivals or timeouts. Scheduling delays are unbounded; however, *most* actual scheduling delays are shorter than a known constant s. When scheduling delays exceed s, servers suffer performance failures [12]. Processors and servers use self-checking mechanisms so it is very unlikely that they produce functionally erroneous outputs. Thus, servers have *crash/performance* failure semantics. Since we are interested in highly available applications, we assume that all crashed servers eventually restart. Lower case letters p, q, r, ... are used to denote both processors and the servers that run on them. Processor names are totally ordered. The previously introduced likely bounds d and s on processor to processor communication and scheduling delays determine a higher level worst-case server to server timeout delay of $\delta = s + d + s$.

We call a distributed system that satisfies the above hypotheses on processors, servers and

[1] As of this writing, versions of the AAS system have been installed in Great Britain and Taiwan, and are scheduled to be deployed in the US by January 1997.

[2] For a discussion on how to decide what "most" means in order to achieve a certain failure semantics, the interested reader is referred to the section "Choosing a Failure Semantics" of [10].

communications a *timed asynchronous* system[3]. Most existing distributed systems are timed asynchronous. Previously (e.g. [6, 10, 15]) we called such systems simply asynchronous, as opposed to the synchronous systems investigated in [12, 9, 7, 11]. This has created confusion, since other authors (e.g. [20]) have used the adjective "asynchronous" with another meaning. The difference comes from the fact that the services of interest to us are timed, while those investigated in [20] are time-free.

Introducing the d and s likely time bounds makes processor and communication service specifications *timed*: they prescribe not only which state transitions/outputs should occur in response to trigger events, such as message arrivals or timeouts, but also the real-time intervals within which they are expected to occur [10]. In contrast, the specifications considered in [20] are *time-free*: they specify, for each state and input, only the next state/output, *without imposing* any constraint on the real-time it takes a state transition/output to occur. Thus, a time-free processor is, by definition, "correct" even when it would take it an arbitrary amount of time (e.g. months or years) to react to an actual input. This very weak definition of correctness makes it impossible for a processor to decide if another processor is correct, crashed, or just slow. A consequence of this weak definition of correctness is the impossibility of implementing fundamental fault-tolerant services such as consensus and membership in time-free asynchronous systems [20, 3]. These services are, however, implementable in timed asynchronous systems in which certain stability conditions hold [16, 19]. Practical systems are often required to be fault-tolerant, so they are naturally timed and make use of timeouts[4]. Thus, most existing distributed systems are timed asynchronous. Since this paper only examines timed asynchronous systems, we will refer to them simply as asynchronous in the following text.

Well-tuned (production) asynchronous systems allow timely communication *most of the time*. However, due to congestion and other adverse phenomena, processors may become temporarily disconnected. When we say that processors p, q are *connected* in a time interval $[t, t']$ we mean that p and q are correct (i.e. non-crashed and timely) in $[t, t']$ and each message sent between them in $[t, t'\text{-}\delta]$ is delivered within δ time units. When we say that p, q are *disconnected* in $[t, t']$, we mean that no message sent between them is delivered in $[t, t']$ or p or q is crashed in $[t, t']$. Processors p, q are *partially connected* in $[t, t']$ when they are neither connected nor disconnected in $[t, t']$. For example, when transient network overload causes some, but not all, messages between p and q to be lost or be late, p and q are partially connected.

Communication between any two processors p and q can only be in one of the above abstract 'prophecy' modes: connected, disconnected or partially connected. These are being

[3] For a formal definition of the timed asynchronous system model, see [14].

[4] While it is true that many of the services encountered in practice do not have explicitly-defined response-time promises, it is also true that all such services become "timed" whenever a higher level service that depends on them, in the worst case the human user, fixes a timeout delay for deciding of their failure.

introduced to allow an external observer to *predict* what happens if messages are actually being sent between p and q, rather than to allow p or q to *evaluate* what their communication mode is at a certain instant (such evaluation is impossible because processes cannot predict the future). Stochastic methods may be used to predict the probability that communication between two processors will be in a certain mode during certain time intervals of interest. Such methods allow then to predict the probability that progress is made in such time intervals.

When we say that an asynchronous system is *stable* in $[t, t']$ we mean that, throughout $[t, t']$: (1) no processor fails or restarts, (2) all pairs of processors in P, are either connected or disconnected, and (3) the 'connected' relation between processors is transitive. Because of the low failure rates achieved with current processor and communication technologies, well-tuned asynchronous systems are likely to alternate between long stability periods and comparatively short instability intervals.

3 Synchronous System Model

Asynchronous systems are characterized by communication *uncertainty*: a server p that tries to communicate with q and times out cannot distinguish between scenarios such as: a) q has crashed, b) q is slow, c) messages from p to q are lost or slow, d) messages from q to p are lost or slow, even though q may be correct and may receive all messages from p.

Synchronous systems relie on real-time diffusion to make communication between correct processors *certain*. Processor p *diffuses* a message to processor q by sending message copies in parallel on all paths between p and q. The implementability of a real-time diffusion service depends on adding the following stronger assumptions to the asynchronous system model discussed before. H1) All communication delays are smaller than δ, all scheduling delays for processes that implement diffusion and broadcast are smaller than s. H2) The number of communication components (processors, links) that can be faulty during any diffusion is bounded by a known constant F. H3) The network possesses enough *redundant* paths between any two processors p, q, so that q always receives a copy of each message diffused by p despite up to F faulty components. H4) The rate at which diffusions are initiated is limited by flow control methods; this rate is smaller than the rate at which processors and servers can correctly receive and process diffusion messages. Methods for implementing real-time diffusion in point-to-point and broadcast networks are discussed in [12, 7, 21], where it is shown that, under assumptions H1-H4, any message m diffused by p is received and processed at q within a computable *network delay* time constant N (which depends on F, network topology, and δ). We say that a communication network is diffusion-synchronous, or simply *synchronous*, if it ensures that any diffusion initiated by a correct processor reaches all correct processors within N time units.

A synchronous network enables processor clocks to be *synchronized* within a known maxi-

mum deviation ϵ. To highlight commonalities between synchronous and asynchronous group communication protocols, this paper will not always distinguish between real-time and synchronized clock time. It is important however to remember that in a synchronous context, such as section 4, time means clock time (as in [12, 9, 7]), while in an asynchronous context, such as section 5, time means real-time (as in [16, 19]), unless otherwise specified. Diffusion and clock synchronization enable the implementation of a synchronous *reliable broadcast* service [12, 7, 21] which, for some constant D (depending on N and ϵ), ensures the following properties: 1) if a processor p starts broadcasting message m at (local) time t, then at (their local) times $t + D$ either all correct processors deliver m or none of them delivers m (atomicity), 2) if p is correct, then all correct processors deliver m at $t + D$ (termination) and 3) only messages broadcast by team members are delivered, and they are delivered at most once (integrity). The processor to processor reliable broadcast defines a new worst-case end-to-end bound for server to server broadcasts of $\Delta = s + D + s$. When one adds to the above broadcast requirements the order requirement that all messages delivered by correct processors be delivered in the same order, one obtains a synchronous *atomic broadcast* service [12, 7]. Since the protocols for synchronous reliable and atomic broadcast are so similar, we will assume that they have the same termination time Δ. For simplicity, we also assume that messages made available for delivery by a broadcast service are consumed *instantaneously* by service users, that is, a message scheduled for delivery to a broadcast user p at time $t + \Delta$ is applied by p *at* $t + \Delta$ (instead of *by* $t + \Delta + s$)[5]. The above "instantaneity" assumption allows us to simplify the description of group communication by ignoring delays added by process structuring (for a more rigorous analysis that takes into account delays between message delivery deadlines and actual message delivery times, the interested reader is referred to [28]).

4 Synchronous Group Communication

We motivate the requirements for synchronous membership and group broadcast informally by making use of the generic service S and team P introduced earlier. We then give the detailed properties used to characterize these two services.

4.1 Motivating membership and group broadcast requirements

The S service exports *queries*, which do not have side effects, and *updates*, which change the service state. Updates are not assumed commutative. The service S is assumed deterministic, that is, its behavior is a function of only the initial S-state s_0 and the updates seen

[5] From an implementation point of view, this can be approximated if the membership, reliable, atomic broadcast and S services are all implemented by the same process in each team processor.

so far. At any moment, the current state of the replicated S implementation is defined by: 1) the *group* of correct S-servers that interpret S-requests and 2) a service-specific S-state, resulting from applying all S-updates issued so far to s_0. Since we are considering a unique service S and team P, we refer for brevity to correct S-servers as *servers* or *group members*; to correctly running team members as *processes*; to S-requests as *requests*; to S-updates as *updates*; and to S-states as *states*. For simplicity, we assume that no total system failures occur.

Since in a synchronous system correct processes are always *connected*, to maximize service availability, a membership service should force all correct processes with up-to-date state replicas to be members of the *current* server group. This not only maximizes availability, but also leads to the greatest potential for load distribution among servers. For example, if $\{p, q, r, s, t\} \subseteq P$ is the membership of the current server group, and client names start with alphabet letters, p could be responsible for replying to service requests from clients whose first letter is in the range a-e, and q, r, s, t could handle ranges f-j, k-o, p-t, and u-z, respectively. If most requests are queries, this reduces processing substantially, since each query is handled by just one server. To maintain consistency, any state update must still be applied by *all* servers, but only one server has to send an acknowledgement to the requesting client.

The simple load distribution policy above assumes in fact that any two group members *agree on the group membership*. Otherwise, some requests might not be processed at all while others might be processed several times. Correct load re-distribution despite failures and joins, can be obtained by achieving a stronger agreement on a *unique order in which failures and joins occur*. To see why this is so, consider that the policy for load re-distribution when a server fails is that its load should be re-distributed among the surviving servers as evenly as possible. For example, in response to p's failure, the load distribution in the new group of surviving servers $\{q, r, s, t\}$ should be: q: a,e-j, r: b,k-o, s: c,p-t, and t: d,u-z. A further failure of r would result in a new group $\{q, s, t\}$ in which the load would be: q: a,b,e-k, s: c,l,m,p-t and t: d,n,o,u-z. A failure of r followed by a failure of p would, of course, result in a very different history of surviving groups and hence, a different final load re-distribution: q: k,l,o,f-j, s: a,b,m,p-t, t: c-e,n,u-z. Thus, if q and t would observe one history of surviving groups $\{q, r, s, t\}$, $\{q, s, t\}$ while s would observe another history $\{p, q, s, t\}$, $\{q, s, t\}$, q and s would both handle requests from clients in range a-b, while requests from clients whose names start with l or o would remain unhandled! A similar example can be given to motivate the need to agree on the order of joins. An earlier paper [9] has proposed that a synchronous *membership service* should achieve agreement on the history of all groups that exist in a system over time and on the membership of each such group. This will obviously allow agreement on the membership differences that exist between successive groups, that is, achieve agreement on a unique history of team member failures and joins.

After joining a group g, different members of g (to be called in what follows g members for brevity) generally receive state update requests at different, unpredictable times. Since

no update commutativity is assumed, to maintain replica consistency all g members need to agree on a *unique order* in which they apply the updates. An earlier paper [12] has proposed that the role of a synchronous *atomic broadcast* service is to achieve agreement on a unique history of updates. To maintain consistency, g members must also agree on the service state s_g when they join g; by a slight abuse of language, we refer to s_g as the *initial g group state*, despite the fact that s_g is in general different from the initial service state s_0. If g_0 is the first group to exist, the initial g_0 state s_{g_0} must be the initial service state s_0. The initial state of future groups is defined inductively as follows. If group g_2 succeeds group g_1, the initial g_2 state s_{g_2} is the final g_1 state s^{g_1}, where the final g_1 state s^{g_1} is the result of applying all updates accepted by g_1 members to the initial g_1 state s_{g_1}. While g_1 members that also join g_2 know the final state s^{g_1} (and hence, the initial g_2 state s_{g_2}) any newly started server p that joins g_2 without having been joined to g_1 must *learn* of the initial group state s_{g_2} by getting it from a member of g_2 that was also a member of g_1. It is convenient to think of such a state transfer to p as being *logically equivalent* to p's learning of the sequence U of all updates accepted in all groups that preceeded g_2 (since the state s_{g_2} that p receives is the result of applying U to s_0).

4.2 Synchronous Membership Properties

New groups of team members are created dynamically in response to server failure and team member start events (for simplicity we do not diferentiate between "voluntary" server departures and "involuntary" failures). At any time, a server can be joined to at most one group. There are times at which a team member may not be joined to any group, for example, between the time it starts and the moment it joins its first group (we assume that a process always initiates a group join request at the same time it starts). All groups that exist over time are uniquely identified by a *group identifier* g drawn from a totally ordered set G. Group identifiers are essential for distinguishing between groups that have the same membership but exist at different times in the history of a system. The membership of any group g is, by definition, a subset of the team P.

The membership service can be specified by defining its state variables and the safety and timeliness properties that it satisfies. Each team member p that is non-crashed maintains the following three membership state variables: $joined$ of type Boolean, $group$ of type G, and mem of type subset of P, with the following meaning: $joined(p)$ is $true$ when p is joined to a group and is $false$ otherwise; when $joined(p)$ is true, $group(p)$ yields the identifier of the group joined by p and $mem(p)$ yields p's local view of the membership of *group(p)*. The values of the above state variables depend not only on their location (i.e the domain P) but also on the point in time at which they are examined (i.e. time domain). However, to simplify our presentation, we leave the time domain of these variables implict in this

presentation[6]. Since we require all members of the same group g to agree on their local view of g's membership, we will sometimes write $mem(g)$ to mean $mem(group(p))$ for some member p joined to g. We say that a group g' is a *successor* of a group g if there exists a member p of g such that the next group p joins after leaving g is g' (p leaves a group as soon as it is no longer joined to it). We denote by $succ(g, p)$ the successor of group g relative to p. Equivalently, when $g' = succ(g, p)$ we say that g is a *predecessor* of group g' relative to p, and we write $g = pred(g', p)$. When $g' = succ(g, p)$, we also say that p *successively* joins groups g and g'.

The membership *interface* consists of a "join-request" downcall and two upcalls. A process calls "join-request" when starting. The first upcall, to a client supplied "state?" procedure, asks for the value of the client's local state (to transfer it, if necessary, to newly started processes, according to the synchronous join protocol of [9]). A process that starts responds to a "state?" upcall by supplying the initial service state s_0. The other upcall, to a "new-group" client supplied procedure, notifies the client (in our case the S-server) that it has just joined a new group g. This upcall has (at least) two parameters supplied by the membership service: $mem(g)$ and s_g, the initial g state. A synchronous membership service M is required to satisfy the following safety and timeliness properties[7]:

(M^s_m) *Agreement on group membership.* If p and q are joined to the same group g, then they agree on its membership: if $joined(p)$ and $joined(q)$ and $group(p)$=$group(q)$ then $mem(p)$=$mem(q)$.

(M^s_r) *Recognition.* A process p joins only groups in which he is recognized as a member: if $joined(p)$ then $p \in mem(p)$.

(M^s_i) *Monotonically increasing group identifiers.* Successive groups have monotonically increasing group identifiers: $g < \mathrm{succ}(g, p)$.

(M^s_a) *Addition justification.* If p joins a group g'=succ(g, p) at time t' such that g' contains a new member q (i.e. $q \in mem(g')$-$mem(g)$), then q must have started before t'.

(M^s_d) *Deletion justification.* If p joins a group g'=succ(g, p) at t' such that a member q of its predecessor group g is no longer in g' (i.e. $q \in mem(g)$-$mem(g')$), then q must have failed before t'.

[6]The time at which the value of these variables is evaluated should be clear from the context, for example property (M^s_m) should be understood as "for all t, t', p, q: if $joined(p)(t)$ and $joined(q)(t')$ and $group(p)(t)$=$group(q)(t')$ then $mem(p)(t)$=$mem(q)(t')$", while property (M^s_r) should be understood as "for all t, p: if $joined(p)(t)$ then $p \in mem(p)(t)$".

[7]When writing service properties we use the notation S^t_p, where S represents the service name, the superscript designates the property type: s for safety and t for timeliness, and the subscript differentiates between different properties of S.

(M_h^s) *Agreement on linear history of groups.* Let p, q be members of a common group g. If p and q stay correct until they join their successor groups g'=succ(g,p) and g"=succ(g,q), respectively, then these successor groups must be the same: $g' = g$".

(M_d^t) *Bounded failure detection.* There exists a time constant D such that if a g member q fails at t, then each g member p correct throughout $I = [t, t+D]$ joins by $t+D$ a new group g' such that $q \notin mem(g')$.

(M_j^t) *Bounded join delay.* There exists a time constant J such that, if p starts at t and stays correct throughout $I = [t, t+J]$, then p joins by $t+J$ a group that is also joined by all processes correct throughout I.

(M_s^s) *Group stability.* If no process failures or joins occur in $[t, t']$, then no server leaves its group in $[t + max(D, J), t']$.

The *synchronous membership protocols* of [9], which depend on the synchronous reliable broadcast specified earlier, satisfy the safety and timeliness properties above. The protocols use local clock times for group identifiers, and ensure lockstep progress, in the sense that all members joining a new group g join it at the same local time $g + \Delta$. The values of the D and J constants for the first protocol of [9] are for example: $\pi + \Delta$ and 2Δ, respectively, where π is the period for broadcasting "I-am-alive" messages. To ensure that servers will not be confused by too close failures and joins, it is sufficient that the delay between a server crash and its restart be at least $max(D, J)$. Any servers p, q that join a common group g agree on a unique subsequent history h of groups for as long as both stay correct (M_h^s). Since, for each group in h, p and q agree on its membership (M_m^s), they agree on a unique order in which failures and joins occur[8]. The timeliness properties (M_d^t,M_j^t) bound the time needed by servers to learn of failures and joins. On the other hand, the safety properties (M_a^s, M_d^s) require that new groups be created *only* in response to failures and joins and (M_j^t) implies that all created groups are maximal. Thus, synchronous membership provides accurate, up-to-date information on which processes are correct and which are not. In particular, the service can be used to implement another frequently needed service, the *highly available leadership* service [9]. A synchronous leadership service is required to ensure: 1) the existence of at most one leader at any point in real time 2) the existence of a real-time constant E such that, if the current leader fails at real-time t, a new leader exists by t+E. To implement this service, it is sufficient that any process that suffers a performance failure at real-time t stops communicating with others past real-time $t + \Delta - \epsilon$[9], and that, for any group g created by the membership service, the member with the smallest identifier play the role of leader. These two leader election rules ensure $E = \pi + \Delta + \epsilon$.

[8] If the memberships of successive groups contains several deletions or additions, these can be ordered following an arbitrary convention, for example by using the total order on team member names.

[9] Such performance failures can be detected and transformed into crashes or requests for re-joining a new group as suggested in [9] by letting a server s process an event e scheduled to be processed by local deadline t only if the local clock value when s is awaken to process e is at most t.

4.3 Synchronous Group Broadcast Properties

A synchronous group broadcast service can be implemented by the members of any group g created by a membership protocol satisfying the previous specification if they add to the atomic broadcast protocols of [12, 7] the following restriction: any update u delivered by a g member p is *applied* by p (to its local state replica) only if the sender of u is a member of g. The resulting *group atomic broadcast* service has the following *interface*. A down-call "broadcast(s,u)" initiates the broadcast of u if the calling server s is joined to a group, otherwise signals an exception. An up-call "update(s,u)" notifies a broadcast service user of the broadcast of u by a member s of the current group. A synchronous group atomic broadcast service B is required to satisfy the following safety and timeliness properties (for the Δ time constant introduced in section 3):

(B_a^s) *Atomicity.* If g member p broadcasts an update u at time t, then either (a) u is applied at $t+\Delta$ by all g members that are correct in $[t, t+\Delta]$, or (b) u is not applied by any g member correct in $[t, t+\Delta]$.

(B_o^s) *Order.* Let p, q be team members that have both applied updates u_1 and u_2. If p has applied u_1 before u_2, then q has also applied u_1 before u_2.

(B_c^s) *Causality.* If u_2 depends causally [24] upon u_1, and u_2 is applied by some correct team member, then u_1 is applied before u_2 by all team members.

(B_t^t) *Termination.* If a correct g member broadcasts u at time t, then u is applied at $t+\Delta$ by all g members correct in [t,t+Δ].

(B_i^s) *Integrity.* Only updates broadcast by a team member joined to a group are applied by team members. Each update is applied at most once.

(B_{ig}^s) *Agreement on initial group states.* Let p be a starting process that joins group g'. If g' has a member q previously joined to g=*pred*(g', q), p's copy of the initial g' state must be set to q's copy of the initial g' state (which must reflect all updates applied by q by the time it joins g'), else p's copy of the initial g' state is set to s_o.

(B_{ud}^s) *Updates precede departures.* If g member p broadcasts u and then fails, any surviving g member that applies u does it before learning of p's failure.

(B_{ju}^s) *Updates follow joins.* If g member p applies an update u broadcast by team member q, then all members of the group g have learned of q's join of g before they apply u.

(B_h^s) *Synchronous agreement on update history.* Let p,q be correct servers joined to a group at time t and let $h_p(t)$, $h_q(t)$ be the histories of updates applied by t by p and q, respectively. Then $h_p(t)$ and $h_q(t)$ are the same.

The above properties imply the following global synchronous group communication property:

(MB_a^s) *Agreement on failures, joins and updates.* If team members p, q, are correct between local time t when they join group g and local time t' when they join group g', then p and q see the same sequence of join, failure and update events in $[t, t']$.

This easy-to-understand property substantially simplifies the programming of replicated applications (see [11] for an example). If applications agree on their initial state and undergo *deterministic* state transitions *only* in response to upcalls to their "new-group" and "update" routines, the total order on joins, failures, and updates observed by correct team members ensures consistency of their states at any point in (synchronized) time.

5 Asynchronous Group Communication

Synchronous group communication simplifies replicated programming considerably, since each replica has the same, accurate, up-to-date knowledge of the system state. However, this comes at a price: the need to ensure that hypotheses H1-H4 hold at run-time. If these hypotheses become false, the properties mentioned previously may be violated.

Asynchronous group communication services can be designed with goals similar to the synchronous goals discussed earlier: G1) agree on a linear history of server groups, G2) for each group g, agree on the initial g state and on a linear history of g updates, and G3) for successive groups g, g', ensure that all members of g' correctly inherit the replicated state maintained by the members of g. However, communication uncertainty introduces a number of complications. First, since processes cannot distinguish between process and communication failures, to achieve agreement on a linear history of groups one has to impose some restriction on the kind of groups that can contribute to history. For example, one could restrict the groups that can contribute to history to be majority groups, where a group g is a *majority group* if its members form a numeric majority of the team members, that is, $|mem(g)| > \frac{|P|}{2}$. This restriction can in fact be used not only to order groups on a history line to achieve (G1), but also to ensure (G3) by relying on the fact that any *two successive majority groups have at least a member in common.* Second, while it is possible to design asynchronous protocols with safety properties similar to those of the synchronous protocols, the timeliness properties satisfied by asynchronous protocols are much weaker: the delays with which processes learn of new updates, joins, and failures are bounded only when certain *stability conditions* hold. The stability condition considered in this paper is *system stability* as defined in section 2. (Weaker stability conditions, such as majority stability and Δ-stability are investigated in [19, 18].) Third, because delays are unbounded in asynchronous systems, ensuring agreement on initial group states requires more work than in the synchronous case.

An earlier paper [16] explored a suite of four increasingly strong asynchronous membership specifications. All protocols described in [16] generate both minority and majority groups, but while the first two expose all these groups to membership service users, the last two restrict the groups visible to users to be majority groups only. For the first (one-round) and the second (three-round with partition detection) protocols, the "successor" relation on the groups seen by membership service users has "branches" and "joins": groups can split and merge. Thus, these two protocols do not construct a linear history of groups. The third (three-round majority) protocol is the first protocol of the suite to achieve agreement on a *linear history of completed majority groups*, where a group is termed *completed* if it is joined by all its members. The "successor" relation for majority groups can still have short lived branches of incomplete majority groups off the main linear branch of completed majority groups. The last (five-round) protocol, achieves agreement on a linear history of all majority groups. Thus, this protocol allows no "branches" in the history at all.

If one were to use the first two membership protocols of [16] and allow state updates to occur in both minority and majority groups to achieve *group agreement* [8], the state views held by team members joined to different groups that co-exist in time could diverge. The three-round with partition detection protocol enables team members to *detect* all potential divergences between the states of merging groups. Once such potential conflicts are detected, the methods of [32] can be used to automatically merge group states, for example when updates are commutative or when only the most recent update to a replicated variable is of importance. However, since for most practical applications, such automatic conflict resolution is not feasible, the price generally paid for allowing updates in minority groups is the need to have manual conflict resolution [30]. If updates are allowed to occur *only* in the completed majority groups created by the three-round majority (or the more expensive five-round) protocol, one can achieve (either majority or strict) agreement on a unique history of updates [8]. *Majority agreement* ensures that all team members currently joined to a completed majority group agree on a unique history of updates; other correct team members not joined to this group might have divergent views on the history of updates. Some applications, however, cannot tolerate any replica state divergence at all [31]. These require the stronger strict agreement. *Strict agreement* guarantees that all correct team members p agree on a linear history h of updates by ensuring that, at any time, any team member p sees a prefix of h. Issues related to achieving partial (that is, group and majority) agreement on update histories are discussed in [8, 16]. In this article we limit ourselves to discuss strict agreement, the asynchronous agreement which resembles most the synchronous agreement presented earlier.

5.1 Agreeing on a Linear History of Completed Majority Groups

As in the synchronous case, all groups created by the membership service are uniquely identified by a *group identifier* g drawn from a totally ordered set G. In our requirements, a universally quantified group g can either be a minority or a majority group. When we

restrict our attention to majority groups we always mention this explicitly. The state of the membership service is defined by the same replicated variables (i.e. *joined*, *group* and *mem*) with the same meanings. An asynchronous membership service M' that achieves agreement on a unique history of completed majority groups should satisfy the (M_m^s), (M_r^s) and (M_i^s) safety properties introduced earlier. The timeliness properties to be satisfied are, however, weaker than in the synchronous case:

(M_d^t)' *Conditionally bounded partition detection delay.* There is a time constant D' such that, if team members p, q are disconnected in $I = [t, t + D']$, the system P is stable in I, and p stays correct throughout I, then p is joined in I to a group g such that $q \notin mem(g)$.

(M_j^t)' *Conditionally bounded join delay.* There is a time constant J' such that, if team members p, q are connected in $I = [t, t + J']$ and the system P is stable in I, then p, q are joined to a common group g in I.

The D' and J' constants provided by the three-round majority protocol of [16] are 9δ + $\max(\pi + (|P| + 3)\delta, \mu)$, where μ is the period for "probing" the network connectivity. The protocol uses three rounds of messages to create a new group g: first the group creator *proposes* g to all team members, second, some of these *accept* to join g, and third, the creator of g defines the membership $mem(g)$ of g as consisting of all accepting team members and then lets all g members effectively *join* g. In addition to the properties mentioned above, this membership protocol also satisfies the following safety properties:

(M_s^s)' *Conditional stability of groups.* If the system P is stable in $[t, t']$, then no server leaves its group in $[t + max(D', J'), t']$.

(M_a^s)' *Justification of additions.* If a process p joins g at t and g has a new member q that was not in the predecessor group $pred(g, p)$ joined by p, then some $r \in mem(g)$, $r \neq q$, was not disconnected from q in $[t', t]$, where t' is the time at which r left its predecessor group $pred(g, r)$.

(M_d^s)' *Justification of deletions.* If p joins g at t and g no longer has a member q that was in the predecessor group $pred(g, p)$ joined by p, then some $r \in mem(g)$ was not connected to q in $[t', t]$, where t' is the time at which r left its predecessor group $pred(g, r)$.

(M_y^s) *Join Synchronization.* If p joins group g at t, then no member of g will be joined to a group $g' < g$ after t.

(M_p^s) *Predecessor Notification.* When p joins majority group g, the membership service notifies p of the majority predecessor group $Pred(g)$ of g, where $Pred(g)$ is the highest majority group $g'<g$ that was joined by a g member.

(M_h^s)' *Agreement on a linear history of completed majority groups.* Let $g < g$' be two completed majority groups. Then g is an ancestor of g', where an *ancestor* of g' is a group g such that either g=$Pred(g')$ or g is an ancestor of $Pred(g')$.

Together with the (M_m^s) property, properties (M_p^s) and (M_h^s)' ensure that all processes *successively joined* to completed majority groups agree on a unique, "official" history of completed majority groups, and hence, on a unique history of joins and failures as seen by the members of these groups. Property (M_p^s) allows any member p of a newly created majority group g to *know* whether it has the correct view of the "official" history by checking whether it is on the "official" history branch. If agreement on a unique order of failures and joins is important, only servers joined to *completed* groups on this official history branch can act on membership changes. While properties (M_d^t)' and (M_j^t)' ensure that there will be no incomplete groups if the system remains stable for at least max(J',D') time units, the following (unconditional) timeliness property bounds the time a process can be joined to an incomplete group, *independently* of whether the system is stable or not:

(M_i^t) *Bounded Incompletness Detection Delay.* There is a constant D" such that, if at time t, p joins a group g with $q \in mem(g)$, p stays correct throughout $I = [t, t + D"]$ and at no time in $I = [t, t + D"]$ q joins g, then p learns of g's incompletness (and leaves g) by t+D".

Properties (M_y^s) and (M_i^t) are useful in implementing an asynchronous *highly available leadership* service, satisfying the following requirements: 1) there is at most one leader at any point in real time, 2) there exists a time constant E' such that, if the system is stable in [t,t+E'] and a majority of processes are connected, then there is a leader by t+E'. In [16] it is shown how to implement this service by adding to the three-round majority membership protocol the following leader designation rule: if g is a majority group, the process with smallest identifier in g becomes leader D" time units after it joins g. This yields the value D'+D" for E' and works because properties (M_y^s) and (M_i^t) ensure that any leader that could have existed in a previously completed majority group $g' < g$ has either left g' or has joined g (and hence, knows about the new leader). This also assumes that any process that suffers a performance failure at real-time t detects this and stops communicating with others past real-time $t + D'+D"$.

The asynchronous membership properties are somewhat harder to understand than the synchronous ones because asynchronous membership is less accurate and up-to-date. There are no more bounds on the time needed to learn of joins and failures and groups can be created even when no team member joins or failures occur during periods of system instability. Finaly, the groups created by an asynchronous membership service reflect the "is connected" rather than the "is correct" physical process reality.

5.2 Agreeing on a Linear History of Updates

Perhaps the strict agreement broadcast protocol easiest to understand is the "two-round" *train* protocol of [8]. In this protocol, for any majority group g, a train of updates circulates among g members according to a fixed cyclic order. A g member that wants to broadcast an update waits for the train, appends the update at the end of the train, lets the train move around twice, and then purges the update from the train. An update u transported by the train is applied by any g member p only when p knows that u is *stable*, that is, p sees u for the second time (An update seen for the first time in the train is *unstable*.) Thus, if the system is stable, $B = (2|P| - 1)\delta$ time units are sufficient for all group members to apply an update. These update broadcast rules ensure that 1) all g members agree on a unique history h of updates, where h reflects the order in which updates *board* the train, 2) any g member p appends an update u to *its* local view of the history h only when p knows that a majority of team members know about u. This ensures that even if system instability forces p to separate from the majority group g, any new majority group g' will have at least a member that knows about u and will ensure that u is appended to the history h.

Let g be a completed majority group joined by members p, q by time t, and assume system stability in $[t, t']$. If p, q agree on the initial g state s_g, the train protocol keeps the local states of p, q *consistent* at any time $t" \in [t, t']$, since each applies between t and $t"$ a *prefix* of the history h of updates that have boarded the train by $t"$. The meaning of "prefix" can be defined as follows. Any (local) history h of updates applied in a group can be understood as being a total mapping from a prefix subset $\{1, \ldots, k\}$ of the natural numbers, for some k$\geq$0, into the set U of possible updates:

h: $\{1, \ldots, k\} \longrightarrow U$.

We say that the local history h_p of updates applied by p is a *prefix* of the local history h_q of updates applied by q, denoted $h_p \prec h_q$, if the domain $dom(h_p)$ of h_p, is included in, or is equal to, the domain $dom(h_q)$ of h_q and for each ordinal in $dom(h_p)$ the histories h_p and h_q agree:

$h_p \prec h_q \equiv dom(h_p) \subseteq dom(h_q)$ and $\forall i \in dom(h_p)$: $h_p(i) = h_q(i)$.

If failures or recoveries result in the creation, by some process c, of a new completed majority group g', the final local states of g members when they leave g will not in general be the same, as in the synchronous case. To ensure agreement on the initial g' state, c can proceed as follows. In its first round of "proposal" messages for the new group g', c piggy-backs a request that all processes p previously joined to g send their state when they left g as well as the fragment h_p of the history of unstable updates seen, but not applied in g. After receiving the states and the h_p unstable history extensions piggy-backed on the second round of "accept" membership messages, c computes the initial g' state by applying to the most recent state received its extension. This initial g' state is then sent to all members of g' piggy-backed on the third round of membership "join" messages. The above initial group

state computation protocol ensures that all g' members agree on the initial g' state, since this state is computed by c and sent to all g' members. It also ensures strict agreement, since any update u that could have been applied by a member of g that is no longer in g' must have been seen by at least one member that is both in g and g', since by definition majority groups have nonempty intersections, Thus, u is applied to the initial g' state, and hence becomes part of the history of applied updates for all members of g'.

In addition to the asynchronous membership properties enumerated earlier, the three round majority membership and the two round train protocols satisfy the broadcast properties (B^s_o), (B^s_c), (B^s_{ud}), (B^s_{ju}) given earlier for the synchronous case. To ensure (B^s_{ud}), any server in a majority group g is given the initial g state before it learns of $mem(g)$ and trains are stamped with the current group id g, so that trains with a group id $g' \neq g$ are not accepted by any g member. Property (B^s_{ju}) is ensured by allowing a majority group member to start a broadcast attempt only after it learns of the group membership. The train protocol also satisfies the following properties (for the B time constant defined earlier):

(B^s_i)' *Integrity.* Only updates broadcast by servers joined to majority groups can be applied (at most once) by team members.

(B^t_t)' *Conditional termination.* If a g member broadcasts u at t, and the system is stable in $I = [t - max(J', D'), B]$, then u is applied by all g members in I.

(B^s_{ig})' *Agreement on initial group state.* Let p join a completed majority group g'. If g' has a member q joined to the preceding completed majority group g, p's copy of the initial g' state is set to q's copy of the initial g' state (which must reflect all updates applied by any member of any past completed majority group), else p's copy of the initial g' state is set to s_o.

(B^s_h)' *Strict agreement on update history.* Let p, q be correct team members and let $h_p(t)$ and $h_q(t)$ be the histories of updates applied by real time t by p and q, respectively. Then either $h_p(t)$ is a prefix of $h_q(t)$ or $h_q(t)$ is a prefix of $h_p(t)$.

If we make the progress assumption[10] that every process be eventually connected to a majority of correct processes for a sufficiently long time and the system be stable during that time, a strict agreement protocol also ensures the following atomicity property:

(SB^s_a) *Atomicity.* If a majority group member g broadcasts u then either (a) u is applied by all team members or (b) u is not applied by any team member.

[10]This progress assumption, which we believe to be quite reasonable in practice for any well-tuned asynchronous system, is similar to the majority-stability assumption of [19].

Even though the asynchronous group communication properties are more difficult to understand than the synchronous ones, they can still substantially contribute to simplifying distributed programming (see for an example [15]), whenever the H1-H4 hypotheses used to render communication certain cannot be guaranteed to be true at run-time.

6 Discussion

Synchronous and asynchronous programming are different system design philosophies, the first assuming that communication is certain, the second assuming that it is not. Communication certainty implies strong, easy to understand safety and timeliness properties which simplify replicated programming substantially. Synchronous programming is natural for hard real-time systems, since it guarantees bounded reaction time to events such as update arrivals, failures or joins. The price is the real-time scheduling and hardware redundancy techniques that need to be used to make the probability of violating the hypotheses H1-H4 at run-time sufficiently small (or equivalently, make the *coverage* of these assumptions be sufficiently high [29]).

Asynchronous programming, based on communication uncertainty, is in fact an umbrella for several programming paradigms that differ in their underlying system models. Examples of such models are the time-free model of [20], the time-free model augmented with various "failure detectors" [4], and the timed model considered implicitly in [6] and named in [16]. We believe that most implemented asynchronous group communication systems are (implicitly) based on variants of the timed asynchronous system model [5, 2, 23, 1, 26, 22, 25, 33, 27, 17]. This model does not assume *anything* about the distribution of communication delays: a correct protocol based on it *always* satisfies its safety invariants, and makes guaranteed progress *whenever* the underlying system satisfies certain stability conditions. This achieves a clean separation between logical correctness properties (that always hold) and stochastic properties that predict the probability that the stability conditions will be true at run-time. Knowledge about delay distributions is only needed when estimating the *probability* that the stability conditions will be true. Asynchronous programming is thus natural for soft real-time systems, that guarantee bounded responses with a certain probability.

This paper has emphasized similarities between synchronous and asynchronous programming by only discussing strict agreement, the kind of asynchronous agreement that is closest to synchronous agreement. In reality the field of asynchronous group communication is vaster, strict agreement being one extreme where all replicas agree and group agreement being the other, where replicas managed by members of different parallel groups can disagree. In general, the stronger the agreement achieved by an asynchronous protocol, the easier the protocol is to understand and use. The price is often a higher message and time complexity. Conversely, the weaker the agreement provided by an asynchronous protocol,

the more difficult is its understanding and usage. Protocols that achieve weak forms of agreement, such as group agreement (also called disconnected or partitionable operation [30, 25]) may even require human intervention to solve the conflicts created by diverging replicas. Group agreement protocols compensate for such user unfriendliness by providing lower message and time complexity and higher update availability.

The tradeoffs possible between synchronous and asynchronous programming, as well as the various possible asynchronous agreement semantics, are not very well understood at present. Work is needed to make these different programming paradigms understandable in a unified framework. This paper was intended as a contribution towards this goal.

Acknowledgment: I would like to thank David Powell, for suggesting the adjective "timed" for the timed asynchronous system model, and Dimiter Avresky, for inviting me to present this paper at the IEEE Workshop on Fault-tolerant and Parallel Distributed Systems, April 15, Honolulu, Hawaii. This research was partially sponsored by IBM and the Air Force Office of Scientific Research.

References

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

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

[3] D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. Technical Report 95-1548, Computer Science Department, Cornell University, Ithaca, New York 14853, October 1995.

[4] T. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing*, pages 147–158, Aug 1992.

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

[6] F. Cristian. Probabilistic clock synchronization. *Distributed Computing*, 3:146–158, 1989. Early version: IBM Research Report, San Jose, RJ 6432, 1988.

[7] F. Cristian. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real Time Systems*, 2:195–212, 1990. Early version: IBM Research Report, San Jose, RJ7203, 1989.

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

[9] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991. Early version: *FTCS-18*, 1988, Kyoto.

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

[11] F. Cristian. Automatic reconfiguration in the presence of failures. *Software Engineering Journal*, pages 53–60, Mar 1993.

[12] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation*, 118:158–179, April 1995. Early version: *FTCS15*, June 1985.

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

[14] F. Cristian and C. Fetzer. Timed asynchronous systems: A formal model. Technical Report CSE95-454, UCSD, 1995. Available via anonymous ftp at cs.ucsd.edu as /pub/team/timedAsynchronousModel.ps.Z.

[15] F. Cristian and S. Mishra. Automatic service availability management in asynchronous distributed systems. In *Proceedings of the Second International Workshop on Configurable Distributed Systems*, Pittsburgh, PA, Mar 1994.

[16] F. Cristian and F. Schmuck. Agreeing on processor-group membership in aynchronous distributed systems. Technical Report CSE95-428, UCSD, 1995. Available via anonymous ftp at cs.ucsd.edu as /pub/team/asyncmembership.ps.Z.

[17] P. Ezhilchelvan, R. Macedo, and S. Shrivastava. Newtop: a fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Systems*, Vancouver, Canada., May 1995.

[18] C. Fetzer and F. Cristian. Fail-awareness in timed asynchronous systems. Technical Report CSE95-453, UCSD, 1995. Available via anonymous ftp at cs.ucsd.edu as /pub/team/failAwareness.ps.Z.

[19] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, Newport Beach, CA, Dec 1995.

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

[21] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed Systems*, pages 97–145. Addison-Wesley, 1993.

[22] F. Jahanian, S. Fakhouri, and R. Rajkumar. Processor group membership protocols: Specification, design and implementation. In *Proc. 12th Symposium on Reliable Distributed Systems*, Oct 1993.

[23] F. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed system. In *Proc. 11th Int. Conf. on Distributed Computing Systems*, pages 882–891, May 1991.

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

[25] D. Malki, Y. Amir, D. Dolev, and S. Kramer. The Transis approach to high availability cluster communication. Technical Report CS94-14, Computer Science Department, The Hebrew University of Jerusalem, Israel, 1994.

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

[27] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. K. Budhia, and C. A. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, this issue.

[28] Z. Ping and J. Hooman. Formal specification and compositional verification of an atomic broadcast protocol. *Real-Time Systems*, 9:119–145, 1995.

[29] D. Powell. Failure mode assumptions and assumption coverage. In *Proc of the 22d Symp. on Fault-Tolerant Computing*, pages 386–395, Boston, MA, Jun 1992.

[30] M. Satyanarayanan, J. Kistler, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4), Apr 1990.

[31] A. Schiper and M. Raynal. From group communication to transactions in distributed systems. *Communications of the ACM*, this issue.

[32] R. Strong, D. Skeen, F. Cristian, and H. Aghili. Handshake protocols. In *Proceedings of the Seventh International Conference on Distributed Computing Systems*, pages 521–528, Berlin, Sep 1987.

[33] R. van Renesse, K. Birman, and T. Hickey. Design and performance of Horus: a lightweight group communication system. TR 94-1442, Cornell Univ, dept. of Computer Science, Aug 1994.

Appendix: Comparison of Group Communication Properties

Properties of Synchronous and Asynchronous Membership

Synchronous Membership		Asynchronous Majority Membership	
M_m^s	*Agreement on group membership*: if p and q are joined to the same group g, then they agree on its membership: if $joined(p)$ and $joined(q)$ and $group(p)$=$group(q)$ then $mem(p)$=$mem(q)$.		
M_r^s	*Recognition*: if $joined(p)$ then $p \in mem(p)$.		
M_i^s	*Monotonically increasing group identifiers*: $g <$ succ(g,p).		
M_a^s	*Addition justification*: If p joins g'=succ(g,p) at t' and $q \in mem(g')$-$mem(g)$, then q has started before t'.	M_a^s'	*Addition justification*: if p joins g'=succ(g,p) at t' and $q \in mem(g')$-$mem(g)$, then $\exists r \in mem(g')$,$r \neq q$, not disconnected from q in $[t,t']$, where t is the time r left pred(g',r).
M_d^s	*Deletion justification*: If p joins g'=succ(g,p) at t' and $q \in mem(g)$-$mem(g')$, then q has failed before t'.	M_d^s'	*Deletion justification*: if p joins g'=succ(g,p) at t' and $q \in mem(g)$-$mem(g')$, then $\exists r \in mem(g')$ not connected to q in $[t,t']$, where t is the time r left pred(g',r).
M_h^s	*Agreement on linear history of groups*: if p and q, initially joined to g, stay correct until they join g'=succ(g,p) and g"=succ(g,q), respectively, then $g' = g$".	M_h^s'	*Agreement on linear history of completed majority groups*: Let $g < g$' be completed majority groups. Then $g \in Ancestor(g')$.
M_y^s	*Join Synchronization*: if process p joins group g at t, then no member of g will be joined to a group $g' < g$ after t.		
		M_p^s	*Predecessor Notification*: When p joins group g, the membership service notifies p of the majority predecessor group Pred(g).
M_d^t	*Bounded failure detection*: $\exists$ time constant D such that if a g member q fails at t, then each g member p correct throughout $I = [t, t+D]$ joins by $t+D$ a new group g' such that $q \notin mem(g')$.	M_d^t'	*Conditionally bounded partition detection*: $\exists$ D' such that if p, q are disconnected in $I = [t, t+D']$, the system P is stable in I, and p is correct throughout I, then p is joined in I to a group g such that $q \notin mem(g)$.
M_j^t	*Bounded join delay*: $\exists$ time constant J such that, if p starts at t and stays correct throughout $I = [t, t+J]$, then p joins by $t+J$ a group that is also joined by all processes correct throughout I.	M_j^t'	*Conditionally bounded join delay*: $\exists$ J' such that, if p, q are connected in $I = [t, t+J']$ and the system P is stable in I, then p, q are joined to a common group g in I.
		M_i^t	*Bounded inconsistency detection*: $\exists$ D" such that, if at time t, p joins a group g such that $q \in mem(g)$, and at no time in $I = [t, t+D"]$ q joins g, then p leaves g by $t+D$".
M_s^s	*Group stability*: if no process failures or joins occur in $[t,t']$, then no server leaves its group in $[t + max(D,J), t']$.	M_s^s'	*Conditional group stability*: if the system P is stable in $[t,t']$, then no server leaves its group in $[t + max(D',J'), t']$.

Properties of Synchronous and Asynchronous Atomic Broadcast

Synchronous Broadcast		Strict Agreement Asynchronous Broadcast	
B_a^s	*Atomicity*: if a g member broadcasts u at t, then either (a) all g members correct in $[t, t+\Delta]$ apply u at $t+\Delta$, or (b) u is not applied by any g member correct in $[t, t+\Delta]$.	$B_a^{s'}$	*Atomicity*: if a g member broadcasts u then either (a) u is applied by all team members or (b) u is not applied by any team member.
B_o^s	*Order*: Let p, q be team members that have both applied updates u_1 and u_2. If p has applied u_1 before u_2, then q has also applied u_1 before u_2.		
B_c^s	*Causality*: if u_2 depends causally upon u_1, and u_2 is applied by some correct team member, then u_1 is applied before u_2 by all team members.		
B_i^s	*Integrity*: only updates broadcast by servers joined to groups can be applied (at most once) by team members.	$B_i^{s'}$	*Integrity*: only updates broadcast by servers joined to majority groups can be applied (at most once) by team members.
B_t^t	*Termination*: if an update u is broadcast by a correct g member at t, then u is applied by all g members correct throughout $[t, t+\Delta]$.	$B_t^{t'}$	*Conditional termination*: if a g member broadcasts u at t, and the system is stable in $I = [t - max(J', D'), B]$, then u is applied by all g members in I.
B_{ig}^s	*Agreement on initial group state*: Let p join g' after starting. If g' has a member q previously joined to $g=pred(g', q)$, p's copy of the initial g' state is set to q's copy of the initial g' state (which must reflect all updates applied by q by the time it joins g'), else p's copy of the initial g' state is set to s_o.	$B_{ig}^{s'}$	*Agreement on initial group state*: Let p join a completed majority group g'. If g' has a member q joined to the preceeding completed majority group g, p's copy of the initial g' state is set to q's copy of the initial g' state (which must reflect all updates applied by any member of any past completed majority group), else p's copy of the initial g' state is set to s_o.
B_{ud}^s	*Updates precede departures*: If g member p broadcasts u and then fails, any surviving g member that applies u does it before learning of p's failure.		
B_{ju}^s	*Updates follow joins*: If g member p applies an update u broadcast by q, then g's members have learned of q's join before applying u.		
B_h^s	*Synchronous agreement on update history*: Let p, q be servers joined to a group and let $h_p(t)$, $h_q(t)$ be the histories of updates applied by local time t by p and q, respectively. Then $h_p(t)$ and $h_q(t)$ are the same.	$B_h^{s'}$	*Strict agreement on update history*: Let p, q be correct team members and let $h_p(t)$ and $h_q(t)$ be the histories of updates applied by real time t by p and q, respectively. Then either $h_p(t)$ is a prefix of $h_q(t)$ or $h_q(t)$ is a prefix of $h_p(t)$.