Flaviu Cristian

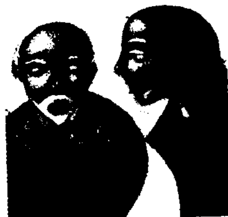
Synchronous and Asynchronous *Group Communication*

How can synchronous and asynchronous group communication services be integrated and understood in a common framework? Experience with group communication in a complex system for air traffic control suggests one viable approach.

In distributed systems, high service availability can be achieved by replicating the service state on multiple server processes. If a server fails, the surviving servers 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 (soft real-time) group communication services can be integrated. This article contributes toward this goal by discussing and comparing the properties that synchronous and asynchronous group communication can provide to simplify replicated programming. The arti-

A consequence of a weak definition of correctness is **the impossibility of implementing fundamental fault-tolerant services.**



cle 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] [12].

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.

Asynchronous System Model

Team processors do not share storage. They exchange messages via a *datagram* communication service. Messages can get lost and communication delays are unbounded, although *most* messages arrive at their destination within a known *timeout* delay constant d [5]. Thus, datagram communication has *omission/performance* failure semantics [9].

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 such trigger events as message arrivals and 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 [11]. 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. For simplicity, we assume that all crashed servers eventually restart. Lower case letters $p, q, r, \ldots$ 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 $\delta = s + d + s$.

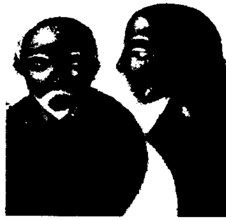
We call a distributed system that satisfies the above hypotheses on processors, servers, and communications a *timed asynchronous* system. Most existing distributed systems are timed asynchronous. Previously (e.g., [5, 9, 13]) we called such systems asynchronous, as opposed to the synchronous systems investigated in [11, 8, 6, 10]. This terminology was confusing, since other authors (e.g., [17]) 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 [17] are time-free.

Introducing the d and s likely time bounds makes processor and communication service specifications *timed,* that is, they prescribe not only which state transitions/outputs should occur in response to trigger events, but also the real-time intervals within which they are expected to occur [9]. In contrast, the specifications considered in [17] are *time-free,* that is, 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. A consequence of this very weak definition of correctness is the impossibility of implementing fundamental fault-tolerant services, such as consensus and membership in time-free asynchronous systems [17, 2]. These services are, however, implementable in timed asynchronous systems in which certain stability conditions hold [14, 16]. Practical systems are often required to be fault-tolerant, so they are naturally timed and make use of timeouts.[2] Since this article examines only timed asynchronous systems, we refer to them simply as asynchronous in the following text.

Asynchronous systems allow timely communication *most of the time*. However, due to congestion and other adverse phenomena, processors may become temporarily discon-

[1]As of this writing, versions of the Advanced Automation Systems have been installed in Great Britain and Taiwan and are scheduled to be deployed in the U.S. by January 1997.

[2]While 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.



In a synchronous context, **time means clock time,** *while in an asynchronous context,* **time means real time.**

nected. When we say that processors p, q are *connected* in a time interval $[t, t']$, we mean that p and q are correct, that is, non-crashed and timely, in $[t, t']$, and each message sent between them in $[t, t'-\delta]$ 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 late, p and q are partially connected. When we say that an asynchronous system is *stable* in $[t, t']$, we mean that, throughout $[t, t']$:

- No processor fails or restarts;
- All pairs of processors in P, are either connected or disconnected; and
- 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.

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 such scenarios as:

- q has crashed;
- q is slow;
- Messages from p to q are lost or slow; and
- Messages from q to p are lost of slow, even though q may be correct and may receive all messages from p.

Synchronous systems rely 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 assumptions to the asynchronous system models already discussed:

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, such as processors and 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 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 [11, 6, 18] 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 maximum deviation ϵ. To highlight commonalities between synchronous and asynchonous group communication protocols, this article does not always distinguish between real-time and synchronized clock time. It is important, however, to remember that in a synchronous context, time means clock time (as in [11, 8, 6]), while in an asynchronous context, time means real time (as in [14, 16]), unless otherwise specified. Diffusion and clock synchronization enable implementation of a synchronous *reliable broadcast* service [11, 6, 18], which, for some constant D (depending on N and ϵ), ensures three properties:

- If a processor p starts broadcasting message m at time t, then at time $t + D$, either all correct processors deliver m or none of them delivers m (atomicity).
- If p is correct, then all correct processors deliver m at

$t + D$ (termination).
- 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 reliable broadcast requirements the order requirement that all messages delivered by correct processors be delivered in the same order, one obtains a synchronous *atomic broadcast* [11, 6]. Since the protocols for synchronous reliable and atomic broadcast are so similar, we assume 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* $t + \Delta + s$.[3]

Synchronous Group Communication

The S service exports *queries,* which do not change the S-state, and *updates,* which change the S-state. Updates are not assumed commutative. The service is assumed deterministic, that is, its behavior is a function of only the initial S-state s_0 and the updates applied so far. At any moment, the current state of the *replicated* S implementation consists of:

- The *group* of correct S-servers that interpret S-requests; and
- 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.

To ensure that servers agree on the state of the replicated service S, it is sufficient that:

G1. they agree on the history of server groups that provide S over time, including the membership of each such group [8]; and

G2. for each group g, all members of g (called g members for brevity) agree on a) the initial service state s_g when g is formed[4] and b) the history of updates to apply to s_g while g exists. [11].

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} (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.

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 differentiate 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 with the same membership but which 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 it satisfies. Each team member p that is non-crashed maintains 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)*. Since we require all members of the same group g to agree on their local view of g's membership, we 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 so 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 [8]). 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

[3]From an implementation point of view, this can be approximated if the membership, reliable, atomic broadcast, and S services are all implemented in a single thread. This "instantaneity" assumption allows us to simplify the description of group communication by ignoring delays caused by multitasking.

[4]Through 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.

satisfy the following safety and timeliness properties[5]:

(M^s_m) *Agreement on group membership.* If p and q are joined to the same group g, 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 it 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, that is, $g < \mathrm{succ}(g, p)$.

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

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

(M^s_h) *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 $g' = \mathrm{succ}(g, p)$ and $g'' = \mathrm{succ}(g, q)$, respectively, then these successor groups must be the same, that is, $g' = g''$.

(M^t_d) *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^t_j) *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 [8], 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 [8] 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^s_h). Since, for each group in h, p and q agree on its membership (M^s_m), they agree on a unique order in which failures and joins occur.[6] The timeliness properties (M^t_d, M^t_j) bound the time needed by servers to learn of failures and joins. On the other hand, the safety properties (M^s_a, M^s_d) require that new groups be created *only* in response to failures and joins, and (M^t_j) 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 [8]. A synchronous leadership service is required to ensure:

- The existence of at most one leader at any point in real time; and
- 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 $t + \Delta - \epsilon$,[7] and that, for any group g created by the membership service, the member with the smallest identifier plays the role of leader. These two leader election rules ensure $E = \pi + \Delta + \epsilon$.

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 [11, 6] 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:

(B^s_a) *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^s_o) *Order.* All updates applied by correct team members are applied in the same order.

(B^s_c) *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^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 $[t, t + \Delta]$.

(B^s_i) *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^s_{ig}) *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'

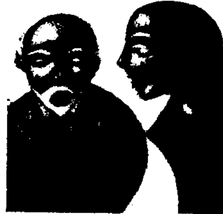
[5]When writing service properties, we use the notation S^t_p, where S represents the service name, the superscript designates the property type—s for safety and t for timeliness—and the subscript differentiates between different properties of S.

[6]If the memberships of successive groups contain several deletions or additions, they can be ordered following an arbitrary convention (e.g., by using the total order on team member names).

[7]Such performance failures can be detected and transformed into crashes or requests for re-joining a new group as suggested in [8] 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 awakened to process e is at most t.

Synchronous group communication simplifies replicated programming considerably,

since each replica has the same, accurate, up-to-date knowledge of the system state.



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

(B^s_h) *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.

These properties imply the following global synchronous group communication property:

(MB^s_a) *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 [10] for an example). If applications agree on their initial state and undergo *deterministic* state transitions in response *only* 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.

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; and

G2. For each group g: a) agree on the initial g state and on a linear history of g updates, and b) 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 be used not only to order groups on a history line to achieve (G1), but also to ensure (G2b) 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 article is *system stability,* as defined in the Asynchronous System Model section. (Weaker stability conditions, such as majority stability, are investigated in [16].) Third, because delays are unbounded in asynchronous systems, ensuring agreement on initial group states requires more work than in the synchronous case.

Another article [14] explored a suite of four increasingly strong asynchronous membership specifications. All protocols described in [14] 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.

If one were to use the first two membership protocols of [14] and allow state updates to occur in both minority and majority groups to achieve *group agreement* [7], the state views held by team members joined to different groups co-existing 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 in [26] can be used to automatically merge group states (e.g., 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 for manual conflict resolution [24]. 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 [7].

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 [23]. 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 [7, 14]. This article will discuss only strict agreement, the asynchronous agreement that most resembles the synchronous agreement presented earlier.

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—*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^s_m), (M^s_r), and (M^s_i) safety properties introduced earlier. However, the timeliness properties to be satisfied are weaker than in the synchronous case:

$(M^t_d)'$ *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 $q \notin mem(g)$.

$(M^t_j)'$ *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 in [14] are $9\delta + \max(\pi + (|P| + 3)\delta, \mu)$, 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 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 $[t + max(D', J'), t']$.

$(M^s_a)'$ *Justification of additions*. If a process p joins g at t and g has a new member q 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 $pred(g, r)$.

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

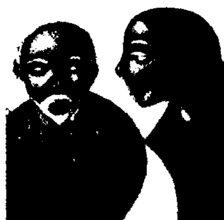
(M^s_p) *Predecessor Notification*. When p joins majority group g, the membership service notifies p of the majority predecessor group $Pred(g)$ of g, where $Pred(g)$ is the highest majority group $g' < g$ that was joined by a g member.

$(M^s_h)'$ *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^s_m) property, properties (M^s_p) and $(M^s_h)'$ 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^s_p) 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

Synchronous and asynchronous programming are different system design philosophies,

the first assuming communication is certain, the second assuming it is not.



branch can act on membership changes. While properties $(M^t_d)'$ and $(M^t_j)'$ ensure there are 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 or is not stable:

(M^t_i) *Bounded Incompleteness 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 incompleteness (and leaves g) by $t + D''$.

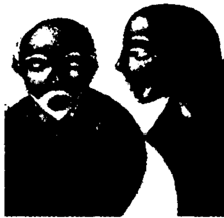
Properties (M^s_y) and (M^t_i) are useful in implementing an asynchronous *highly available leadership* service, satisfying two requirements: 1) There is at most one leader at any point in real time; and 2) There exists a time constant E' such that, if the system is stable in $[t, t + E']$ and a majority of processes are connected, then there is a leader by $t + E'$. In [14], 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 the smallest identifier in g becomes leader D'' time units after it joins g. This leader election rule yields the value $D' + D''$ for E' and works because properties (M^s_y) and (M^t_i) 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. The leader election rule also assumes that any process that suffers a performance failure at real-time t detects it and stops communicating with others past real-time $t + D' + D''$.

The asynchronous membership properties are somewhat harder to understand than the synchronous membership properties 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. Finally, the groups created by an asynchronous membership service reflect the "is connected" rather than the "is correct" physical process reality.

Agreeing on a Linear History of Updates

Perhaps the strict agreement broadcast protocol easiest to understand is the "two-round" *train* protocol of [7]. 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 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*:) This ensures: 1) that all g members agree on a unique history h of updates, where h reflects the order in which the updates proposed by g members *board* the train; and 2) that 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 waiting rule ensures that even if system instability forces p to separate from the majority group g, any new majority group g' will have at least one 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 and 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''. If failures or recoveries result in the creation (by some process c) of a new completed majority group g', the final local states of q members (when they leave g) will not in general be identical, 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 piggybacks 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 piggybacked on the second round of "accept" membership messages, c computes the initial g' state by applying to the most recent state received the corresponding extension. This initial g' state is then



Protocols that achieve weak forms of agreement, *may even require human intervention to solve the conflicts created by diverging replicas.*

sent to all members of g' piggybacked on the third round of "join" membership messages. This 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 no longer in g' must have been seen by at least one member 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 so far, the three-round majority membership and the two-round train protocols satisfy the broadcast properties (B^s_o), (B^s_c), (B^s_{ud}), and (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 broadcasting only after it learns of the group membership. The train protocol also satisfies the following properties:

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

$(B^t_l)'$ *Conditional termination.* If a g member broadcasts u at t, and the system is stable in $I = [t - max(J', D'), t + (2|P| - 1)\delta]$, 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 every process that is correct is eventually connected to a majority of correct processes for a sufficiently long time while the system is stable, the train protocol also satisfies the following atomicity property:

$(B^s_a)'$ *Atomicity.* If a completed 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.

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 [13]), whenever the H1-H4 hypotheses used to render communication certain cannot be guaranteed to be true at run time.

Conclusions

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

Asynchronous programming—based on communication uncertainty—is an umbrella for several programming paradigms that differ in their underlying system models. Examples of such models are the time-free model in [17], the time-free model augmented with various "failure detectors" in [3], and the timed-model considered implicitly in [5] and named in [14]. We believe that most implemented asynchronous group communication systems are (implicitly) based on variants of the timed asynchronous system model (see, for example, Totem, Transis, and Horus in this special section and several other systems [4, 20, 1, 21, 19, 15]). 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. Therefore the timed model 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 needed only 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 article emphasized similarities between synchronous and asynchronous programming by discussing only strict agreement—the kind of asynchronous agreement 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 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 it is to understand and use the protocol. The price is often higher message and time complexity. Conversely, the weaker the agreement provided by an asynchronous protocol, the more difficult it is to understand and use it. Protocols that achieve weak forms of agreement, such as group agreement (also called partitionable operation), may even require human intervention to solve the conflicts created by diverging replicas [22]. Group agreement protocols compensate for such user unfriendliness by providing lower message and time complexity and maximum 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 article is intended as a contribution toward this goal.

Acknowledgments

I would like to thank David Powell for inviting me to write this article and for suggesting the adjective "timed" for the timed asynchronous system model. This research was partly sponsored by IBM and the Air Force Office of Scientific Research.

References

1. Birman, K., Schiper, A., and Stephenson, P. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems 9*, 3 (Aug. 1991) 272–314.
2. Chandra, D., Hadzilacos, V., Toueg, S., and Charon-Bost, B. On the impossibility of group membership. Technical Report 95-1548, Computer Science Dept., Cornell University, Oct. 1995.
3. Chandra, T., Hadzilacos, V., and Toueg, S. The weakest failure detector for solving consensus. In *Proceedings of the 11th ACM Symposium on Principles of Distributed Computing* (Aug. 1992) pp. 147–158.
4. Chang, J., and Maxemchuk, N. Reliable broadcast protocols. *ACM Transactions on Computer System 2*, 3 (Aug. 1984) 251–273.
5. Cristian, F. Probabilistic clock synchronization. *Distributed Computing 3* (1989) 146–158. Early version: IBM Research Report, San Jose, RJ 6432, 1988.
6. Cristian, F. Synchronous atomic broadcast for redundant broadcast channels. *The Journal of Real Time Systems 2* (1990) 195–212. Early version: IBM Research Report, San Jose, RJ 7203, 1989.
7. Cristian, F. Asynchronous atomic broadcast. *IBM Technical Disclosure Bulletin 33*, 9 (Feb. 1991) 115–116. Presented at the *1st IEEE Workshop on Management of Replicated Data* (Nov. 1990, Houston, Tex.)
8. Cristian, F. Reaching agreement on processor-group membership in synchronous distributed systesm. *Distributed Computing 4* (1991) 175–187. Early version: FTCS-18, 1988, Kyoto, Japan.
9. Cristian, F. Understanding fault-tolerant distributed systems. *Commun. ACM 34*, 2 (Feb. 1991) 56–78.
10. Cristian, F. Automatic reconfiguration in the presence of failures. *Software Engineering Journal* (Mar. 1993) 53–60.
11. Cristian, F., Aghili, H., Strong, R., and Dolev, D. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation 118* (Apr. 1995) 158–179. Early version: FICS15, June 1985.
12. Cristian, F., Dancey, B., and Dehn, J. Fault-tolerance in the advanced automation system. In *Proceedings of the 20th Symposium on Fault-Tolerant Computing* (June 1990, Newcastle-upon-Tyne, UK) pp. 6–17.
13. Cristian, F., and Mishra, S. Automatic service availability management in asynchronous distributed systems. In *Proceedings of the 2nd International Workshop on Configurable Distributed Systems*, (Mar. 1994, Pittsburgh, Penn.)
14. Cristian, F., and Schmuck, F. Agreeing on processor-group membership in asynchronous distributed systems. Technical Report CSE95-428, UCSD, 1995. Available via anonymous ftp at cs.ucsd.edu as /pub/team/asyncmembership.ps.Z.
15. Ezhilchelvan, P., Macedo, R., and Shrivastava, S. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Systems* (May 1995, Vancouver, Canada).
16. Fetzer, C., and Cristian, F. On the possibility of consensus in asynchronous systems. In *1995 Pacific Rim International Symposium on Fault-Tolerant Systems* (Dec. 1995, Newport Beach, Calif.)
17. Fischer, M.J., Lynch, N.A., and Paterson, M.S. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32*, 2 (Apr. 1985) 374–382.
18. Hadzilacos V., and Toueg, S. Fault-tolerant broadcasts and related problems. In S. Mullender, ed., *Distributed Systems*, Addison-Wesley, Reading, Mass., 1993.
19. Jahanian, F., Fakhouri, S., and Rajkumar, R. Processor group membership protocols: Specification, design and implementation. In *Proceedings of the 12th Symposium on Reliable Distributed Systems* (Oct. 1993).
20. Kaashoek, F., and Tanenbaum, A. Group communication in the Amoeba distributed system. In *Proceedings of the 11th International Conference on Distributed Computing Systems*. (May 1991) pp. 882–891.
21. Mishra, S., Peterson, L., and Schlichting, R. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed Systems Engineering Journal* (1993).
22. Satyanarayanan, M., Kistler, J., Kumar, P., Okasaki, M., Siegel, E., and Steere, D. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers 39*, 4 (April 1990).
23. Schiper A., and Raynal, M. From group communication to transactions in distributed systems. *Commun. ACM 39*, 4 (April 1996).
24. Strong, R., Skeen, D., Cristian, F., and Aghili, H. Handshake protocols. In *Proceedings of the 7th International Conference on Distributed Computing Systems* (Sept. 1987, Berlin, Germany) pp. 521–528.

About the Author:

FLAVIU CRISTIAN is Professor of Computer Science and Engineering at the University of California, San Diego. **Author's Present Address:** Dept. of Computer Science and Engineering, 0114, University of California, San Diego, La Jolla, CA 92093-0114; email: flaviu@cs.ucsd.edu