Fault Handling in Distributed Systems with Group Membership Services.

Francesc D. Muñoz-Escoí José M. Bernabéu-Aubán Pablo Galdámez-Saiz

Technical Report ITI-ITE-98/5

Abstract

In order to build single system image clusters, it is necessary to provide some sort of fault handling mechanisms, allowing system services to react appropriately to system faults, offering some predetermined degree of high availability. At the base of any fault handling protocols, there must be fault detection mechanisms, indicating when some machine or process fails. Also, at the base of any high availability scheme, there must be some sort of replication. Group Membership Protocols provide two basic types of services: fault detection, and group membership configuration. In addition, some group membership protocols provide also a communications toolkit to facilitate the exchange of information among the members of a related group of processes.

Not all group membership protocols are suitable for any application environment. In fact most of the protocols proposed are tailored for the specific needs of their target systems. This paper discusses the functionality available in the different group membership protocols, and how it can be used to build robust systems.

1 Introduction

In order to achieve robust services in distributed applications it is necessary to replicate the processes which make up such services. Thus, some processes tend to constitute process *groups*, where they have several common attributes which defi ne their state or behavior. Similarly, the group of nodes in a tightly coupled distributed system (like a cluster), also can be seen as a group of processes with common characteristics.

A process group is, therefore, a set of somewhat related processes; either providing the same interface of operations being served or sharing a common state that can be queried from any of the processes or being one of the modules of a distributed application. The same can be applied to a group of computing nodes: it is composed of a set of machines and these machines share some kind of relationship.

A *group membership service* manages the concept of a group, either built using a set of processes or a set of machines. It has to keep the current state of the group, knowing about each possible member of the whole set and responding to any request made by any member about the state of any other, or to requests to join or leave the group.

However, other requirements have been made to these services. Currently a group service has to deal with detecting automatically the failure of any member, and it has to react to this situation, i.e. reconfi guring the group membership, as soon as possible. Other activities that are being included in the tasks to be done by the group service are intragroup communication and the reordering and delivery of messages and notifi cations sent to the group members. So, the membership service forms a base that is being completed by other related services, building a *group communication service* or *toolkit* which usually emphasizes on message delivery order and semantics.

Besides the group communication toolkits, which have been the most common user of group membership services, other software components require service about group membership. One of the emerging clients of this kind of services are the kernels used in multicomputer systems which offer a single system image. These software elements need a membership service to know which machines of the cluster are up and running, so they can do load balancing according to the current confi guration of the system. Moreover, when a failure occurs, the system has to be reconfi gured as soon as possible, and all the machines have to know which is the current group set. If the operating system provides support for highly-available applications or servers, where these software components are replicated, additional support has to be provided by this operating system to detect when one of the components of an application has failed, notifying to the rest about the failure.

Other clients of group services are fault-tolerant distributed applications which use the group services to know the current state of the application's processes. In this case, the group membership service is one of the blocks that has been used to build the application and the services provided are tailored to the application needs.

Independently of the client which uses the group membership services, a service of this kind has to perform the following tasks. First, it has to keep the current membership set of the group. To achieve this, the service has to manage a set of monitor processes in each of the machines where one of the group members may stay. We refer to this software element as a *membership monitor*. All membership monitors have to communicate periodically to know that each other is alive. As the group members are usually placed in different machines, the communication among membership monitors must be done exchanging messages. So, the protocol used to exchange information depends on the kind of distributed system where the group members reside. Thus, the degree of synchrony, the type of interconnecting network, the possibility of network partitions and other characteristics of the system are conditioning the type of protocol to use. In any case, the resulting protocol has to provide to the service the means to know which is the current set of group members.

Support for group membership changes constitute the second set of services provided. When a change occurs in the group, the group must be warned about the current change being done. A change may be caused either by a join of a new member, the departure of one of the current members or by the failure of one of the members. The most difficult change to manage is the last one, due to the kind of communication maintained among the members. Note that several classes of failure may happen in a distributed system (for instance, a machine crash, an omission failure[1], a performance failure[2] or an interconnecting link failure) which may lead the membership service to confusion, because part of them may be indistinguishable from an actual member failure.

Finally, group members may request information about the current membership set. This information is needed when a member must broadcast a message to the rest of the group.

As we have sketched above, the services related to the group membership problem depend mainly on the client application which uses these services and also on the characteristics of the distributed system where the group application runs. Some of the characteristics which define a membership service are outlined in the following sections, and some examples of group membership protocols are also described.

The rest of the paper is organized as follows. Section 2 outlines several aspects that must be taken into account when a new membership protocol is being designed. A brief description of each algorithm property is given and several design alternatives are explained. Section 3 classifies some of the current protocols according to the alternatives previously shown. Section 4 explores some of the membership protocol alternatives that have not been used until now. Possible target systems and applications for these new alternatives are given. Finally, section 5 gives a brief summary of the protocols described in the paper.

2 Characterization of a Membership Service

Several design decisions have to be taken when a new membership protocol is being built. The alternatives chosen determine the properties offered by the protocol to its applications and the kind of system where the protocol can be run. We outline some of these design choices in the following subsections.

2.1 Relationship with Other System Components

Usually, the group membership service is provided by a software module at user level which uses unreliable *communication services* provided by the underlying operating system. Each membership monitor reports any change in the group view to its upper module, the *client application*.

This scheme is depicted in figure 1 and has been used in a large number of group communication toolkits, such as Isis [8, 27], Totem [1, 24] and Transis [11].

The relationship between the membership monitor and the target application can be more or less complex. The minimum service that has to be provided is the notification of membership changes to the application. But this scheme forces the application to send its messages to other members bypassing the membership layer.

[1] An omission failure happens when a message is never delivered to its destination, either due to a buffer overflow in the receiving side or due to an expected message that never was sent.

[2] A performance failure means that a message was not delivered on time, and this is only applicable to systems that set an upper bound on message delivery.

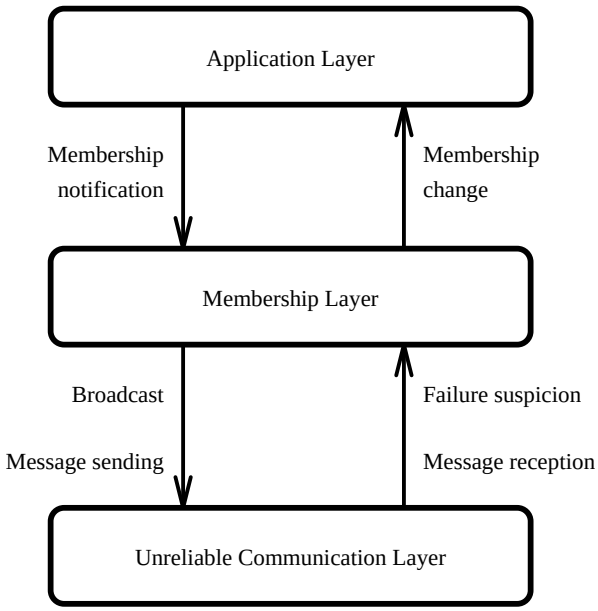

Figure 1: Main system components.

The choice found in group communication toolkits is to integrate the management of the group view with some additional services such as the sending and delivery of messages among the members. The application can choose not to worry about the current group view, it has only to send and receive intragroup messages using the services provided by the group toolkit. So, a group communication tool extends the group membership service with a group communication service which offers some group event ordering semantics.

If a micro-kernel or object-oriented operating system approach is used, the group membership protocol can be included as another component of the operating system [5, 17]. This allows membership services to be provided as common system services, available to all its applications, although it forces to maintain the group view in a per machine basis instead of in a per process one; i.e., the elements of the group must be machines, not processes. This approach may be used in distributed system kernels, which are able to manage the set of machines which constitute the current system.

2.2 Synchrony

Distributed systems are usually asynchronous. Each computing node has a private clock and interprocess communication is done exchanging messages, which can be arbitrarily delayed. Therefore, different nodes have a different notion of time and may work at different speeds. Programming distributed algorithms in this environment is harder than in a synchronous one, where all nodes share the same clock and make progress simultaneously. Moreover, some theoretical results [9] prove that the membership problem can not be solved in asynchronous systems where computing nodes may crash, even if communication channels are reliable and computing nodes are forced to crash by the rest of the group when suspected faulty.

The impossibility result of [9] asserts that in asynchronous environments where only a primary group is allowed (i.e., partitions are not supported,) it is impossible to achieve agreement on the new group view to be installed. In those systems, if a node crashes some nodes are unable to install new group views because more than one proposal for the new view exists. This situation makes the membership service completely useless, since current members can not decide which is the current group view. To overcome the problems offered by asynchronous systems, some degree of synchrony is introduced in the membership protocols.

The first approach is to assume a synchronous system. In [10] the communication subsystem provides two services: an unreliable datagram service and a diffusion service. The diffusion service has a known maximum delay and its messages are always delivered on time. To guarantee this property, sufficient buffer capacity, appropriate diffusion rates and intercommunication link redundancy are assumed. The diffusion service allows clock synchronization, since periodical diffusions can be used to adjust local clocks to a known global time. With little effort, the

diffusion service can be extended in synchronous environments to guarantee an atomic broadcast communication service. As a result, the membership protocols running in this environment can be implemented easily because all the nodes initiate the same steps and expect the same events at the same time.

Another synchronous protocol is shown in [19]. An external clock synchronization protocol is executed apart from the membership one. Since no reliable communication service is assumed, this work introduces the notion of *sparse time* to guarantee the minimum properties needed to develop synchronous algorithms. A sparse time base is a global notion of time with limited precision. This precision is low enough to guarantee that the maximum difference between local clocks is lower than the time unit being used. Its main property is that each system event is viewed by any member of the system at the same time stamp. So, any node in the system drives its membership monitor synchronously with the rest of the group.

Systems like Totem [1, 24], Relacs [2], Isis [8], Transis [11], Horus [31] and others [16, 22, 25, 26] have different approaches to build the new membership set when failures or joins are detected in their asynchronous environments.

Isis is an example of an asynchronous membership protocol that does not tolerate partitions. In its protocol, the oldest group member is the manager of the group and decides when a new view has to be installed. This system uses sequence numbers to distinguish successive group views and to reject stale messages. So, machines suspected faulty are generally forced to crash, because they are isolated by the rest of the group and their messages are ignored. Nevertheless, when the manager is erroneously suspected faulty by part of the group, problems will arise because this situation leads to a partition and the minor subgroup is forced to crash. Frequent partitions may lead to a crash of the whole system when the number of live nodes is too small. This situation may arise in any asynchronous system which only supports a primary group when partitions occur (as suggested by the impossibility result described above,) but fortunately it seldom happens.

Another kind of protocols for asynchronous environments support isolated multiple-member subgroups when a partition arises and a sub-protocol to re-merge these subgroups when the failure disappears [2, 11, 20, 31]. These protocols avoid the situation described in the impossibility result, since the multiple proposals for the next group view are accepted by different members and multiple subgroups are installed. When several subgroups can communicate again among them, they are re-merged in a bigger group view. However, not all highly available applications may afford group splitting. So, this kind of protocols are useless in some environments. A replicated server which has its state distributed among its members (a database server, for instance) may provide inconsistent replies to its clients requests if the group becomes partitioned.

On the other hand, [12] presents some applications which need a membership service that tolerates partitions. CoRel is one of these applications. It is a replication service which is improved using partitionable group support. As a result of a partition, CoRel allows any subgroup to proceed, although only the majority subgroup may perform updates to the replicated state. The other subgroups will know about the updates when they re-merge with the main subgroup. The advantage over a primary-partition approach is that a partitionable service survives situations where the group has been divided into multiple subgroups where none of them reaches the minimum required to proceed; i.e., these situations lead to a crash of the entire group if the primary-partition alternative is being followed, losing the group's state, while in a partitionable service different subgroups will eventually merge and their states are somehow rebuilt.[3]

Summarizing, synchrony is a property that makes easier the task of designing membership protocols, since all the membership servers may take a decision (about a membership change) in a message round and this decision is known by the rest of the group on the following message round. Asynchronous systems are harder to manage since messages may be arbitrarily delayed and failure detection is not accurate.

2.3 Accuracy, Liveness and Safety

One of the main targets of a membership service is to deal with machine failures and joins (membership changes, on the sequel.) To detect these events, the membership service has to use a failure reporting mechanism; i.e., some software component which scans the network and reports the membership server about machine crashes. This failure reporting modules can be easily extended to notify the membership server about the recovery of temporary faulty members or the join of new ones.

The failure reporting mechanism used by a membership service is defi ned by three properties, according to [7]:

[3]However, no comment is given about how minority subgroups deal with non-updating requests. Note that these requests are problematic since they may provide an outdated and inconsistent image of the group state.

Accuracy A membership service is *accurate* if it only reports real membership changes to the target application. In other words, all membership changes must happen before being reported, but there may be changes never reported.

Liveness A membership service is *live* if all membership changes are eventually reported to the application. Note, that the service may report changes that never happened, but all the real changes must be reported.

A stronger property is *bounded liveness* [15], where every change must be detected and reported in a known bounded time.

Safety A membership service is *safe* if at a given time, all operational processes (members of the group) agree on the current membership set, and processes supposed faulty can not communicate with operational ones.

Safety is discussed in greater detail in section 2.10, where we describe how the members of a group can achieve agreement and what sorts of agreement may be attained.

Synchronous systems allow failure detectors to be accurate and live, while asynchronous systems can not guarantee both properties simultaneously.

Assuming a reliable interconnecting network [10], synchronous systems may be accurate. All processors in the system share a common clock, and periodical messages are broadcast and expected by any member. When a message is lost, its source is considered faulty and actually it is. Liveness is also satisfied, since all membership changes are known in the next message round.

In [19] a synchronous system is described, where all members have a synchronized clock and the interconnecting network has a known upper bound delivering messages. Despite this, its protocol does not assume that this network is reliable — links may be temporarily broken or all messages traversing a link may be corrupted — and therefore, accuracy can not be guaranteed, since link and node failures are indistinguishable and the failure reporting service may notify a node failure when only the link has failed.

Delayed messages, interconnecting link failures and machine crashes are indistinguishable in asynchronous systems, since the receiver machine observes the same situation: no message sent by a given machine is delivered in a predetermined period of time. To guarantee accuracy, failure detectors in asynchronous systems have to report membership changes once the event has taken place and the affected machine is running again. Liveness is easier to achieve. Once a failure reporting module suspects another machine is faulty, it reports the change immediately. Therefore, liveness is guaranteed since every change is reported in a bounded time — using failure detectors based on timeouts — and accuracy is lost — since part of the reported changes are not true. Usually, an asynchronous system is accurate to report join events, since a new member is only considered when its messages are received, and they can be delayed or lost an unbounded number of times, and live to report failures, since omission, performance or link failure may be mistakenly considered as node crashes, and when the node actually fails the failure is easily detected. Samples of this kind of systems are Isis [27], Horus [31], Transis [11], Totem [1, 18] and many others.

The IPC provided by the Mach micro-kernel is an example of a failure reporting mechanism that is accurate but not live. When an external process crashes, the local processes which have to communicate with it must wait. The failure of the external process is reported when it is recovered. This approach guarantees accuracy, because only true failures are notified, but is not live since machines or processes which do not recover prevent other intercommunicating processes to advance.

Generally, liveness is preferred to accuracy in asynchronous systems. The lack of accuracy can be partially compensated if safety is guaranteed — for instance, preventing assumed-faulty nodes, even if they are actually operational, to communicate with the current members, — but the lack of liveness may prevent progress since faulty machines are still considered operational.

2.4 Member Classification

Membership services may differ in what they consider a valid member of the group and in the kinds of members being managed. Cristian [10] identifies two membership problems that have to be solved by a membership service:

- *Processor-group membership problem*. To solve it, the membership service must find the current group of machines which compose the group view.
- *Server-group membership problem*. Once the processor-group membership set is found, the service has to know which servers running in those processors belong to the group.

Usually, only one server runs in one processor and a processor is considered alive if it has a membership monitor running on it and maintaining a state which agrees with other servers. Some membership services deal with the machines which compose the group and are only interested in those machines. Their clients are replicated or somehow related applications which run on the system and want to know the state of their remote companions' machines. So, this kind of services consider machines or processors as the members to be controlled. Other membership services deal directly with the replicated applications and consider these elements as their members. Therefore, if two instances of the target application run on the same machine, they are managed as two different members.

Group communication toolkits typically need to solve the server-group membership problem. In Relacs [4] and Phoenix [20] the classes of members that have to be managed by a membership service are extended to three:

- *Core members*. The group consists of a set of core members which provide some kind of service and share their state. This has been the traditional target of regular membership protocols.
- *Clients*. When a process needs to communicate frequently with the core members of a group, it joins the group as a client. Clients do not belong to the main group (in fact, this group only consists of core members) but they are reported about any group change; i.e, about joins and departures of core members that imply a group view change. To allow this, the group membership service receives *explicit requests* to join a group as a client and to leave it. Moreover, it does not have to worry about changes in the client set, since they are not relevant to the normal function of the core members group.

  The use of registered client processes may facilitate the ordering of client requests to the core group when this condition is needed.
- *Sinks*. Sinks processes are a special class of clients which only receive the information reported by core members, but they do not request any other group service. The life of a sink member begins when it requests to join the group as a sink. Hereafter, the sink member receives a sequence of output messages from the core members, until it decides to leave the group. In this case, the sink process is not interested on group changes, it only wants to receive output data from the core group. For instance, a client of a replicated news server can be implemented as a sink for the group that implements that server.

Observe that core members, clients and sinks are processes and we can find some instances of each class on the same machine. Therefore, a group communication system of this kind needs a membership service dealing with processes, since machine granularity is not enough. Besides that, several group views are needed (In particular, one per member class.)

2.5 Event Ordering

Groups deal with different types of events which must follow a given order in all of their members. There are multiple possible orders to apply to the chain of events that arrive to a particular group member, each one offering different guarantees to the programmer who has to use the group as a tool to develop replicated applications.

To be able to describe these event orders, some care must be taken about which are the sources of the events to be considered. Usually, these sources are the emission or reception of messages. But the messages used by a group belong to several classes. To begin with, there are *poll messages* used by the membership monitors to implement the membership protocol. They receive this name because one of the membership protocol functions is to check the state of all other members of the group. The second class consists of *regular messages*. A regular message is transmitted between two group members or between an external process and a group member. Anyway, it is a message that is not generated by the membership protocol. Finally, a *change message* is a message generated by the membership protocol which informs the group members about a change in the current group view. It forces the installation of another group view.

Usually, poll messages are not considered as relevant events. Event ordering is only needed to provide some guarantees about how the group state is updated and poll messages do not introduce any update in the application-level state of a member. However, change messages may be important, since the group members may have to react in some way to the addition or departure of one or more members. Regular messages are also the source of changes in the state of any member, since they are requesting some service from the replica or are providing the answer to a previous request. So, only regular and change messages can be considered sources of group-related events. Moreover, given a particular message, there are some events related to it:

- *Message sending: send(m).* A point-to-point message is sent by a process to some other.
- *Message diffusion: multicast(m).* A message is broadcast or multicast to all the members of the group.
- *Message reception: receive(m).* A regular message is received by the communications layer of its destination's machine, but its contents are not still delivered.
- *Message delivery: deliver(m).* A previously received message is delivered to the client application. This is done when all the group members agree on delivery order.

We are only interested in message delivery events in this section. The different orderings followed by current membership protocols are outlined below, starting with the simpler and more economic ones.

2.5.1 FIFO Ordering of Membership Changes

FIFO order is achieved when messages about the current state of a particular member are delivered to the client applications at all the other members in the same order. To guarantee this property, the same sequence of membership change events regarding a particular node can be found in the history of events of all group members. Note that FIFO order is not respected if some membership changes about a given member are notified in opposite order in at least two different group members.

Usually, FIFO ordering is maintained in all membership protocols.

2.5.2 Total Ordering of Membership Changes

Total order requires that any membership change message, independently of the member who originated such a message, was delivered in the same order at any group member. Therefore, the history of membership change events has to be the same in any member. Note that total ordering only orders membership changes. Nothing is said about regular messages and their relationship with membership changes.

The difference between FIFO order and total order is that the first only requires that membership changes originated by a member must be ordered the same way by the rest of the group, while total order requires that all membership changes must follow the same order in all of the group members. So, in FIFO order if we have three events e_1, e_2, e_3 regarding a member E and three events f_1, f_2, f_3 originated by a member F, if any group member sees these events in the order e_1, e_2, e_3 and f_1, f_2, f_3, independently of how they are placed in the global order, the resulting sequence of events follows FIFO order. On the other hand, if a member orders them in the sequence: e_1, f_1, f_2, e_2, e_3, f_3, then all other members must agree on this sequence to follow total order.

This event ordering is followed by a greater part of the membership protocols that do not allow group partitions. If a group view is allowed to be partitioned and re-merged at a later time, its members do not follow a total order since membership-related events were different while the subgroups had been partitioned. Nevertheless, if we only consider the members of an isolated subgroup, the events seen in that subgroup can be totally ordered. So, the total ordering is only violated when several subgroups join again.

2.5.3 Agreement on Last Message

Besides ordering membership change messages, regular messages (i.e., messages sent by the client application rather than being sent by the membership monitor) exchanged among members must be ordered to avoid inconsistencies in the group state (specially, if the membership set is used to maintain a replicated process.)

Agreement on last message consists in requiring that all members deliver and agree on the same last regular message sent by a site that is currently considered faulty. So, this agreed upon regular message is the last one that precedes the membership change message which announces the failure of that node. There may be additional messages sent by the faulty node before failing. These messages are not considered its last message because some nodes have never received them and, obviously, those nodes will not agree with the rest on considering one of these additional messages as the last one. So, the last message is the last sent message that is received by all the rest of the group and is delivered to any group member.

Similar to how the agreement on last message has been defined for faulty members, the *agreement on first message* can be defined for joining members. Note that a joining member may believe prematurely that it has joined the group and may start to send messages to the group. Messages sent by the joining process will be

delivered when the rest of the group agree on its identity and inclusion in the group. So, some of the first messages of the joining process may be discarded by the current group and it is important that all group members agree on the first message that has to be accepted and delivered.

2.5.4 Virtual Synchrony

Virtual synchrony [8] guarantees that processes perceive member failures or joins at the same logical time. This logical time is increased each time a new event is triggered in the group. Relevant events are message sending and delivery and any group view change. So, to maintain the same logical time on each group member, every process has to hold the same event history.

Note that to guarantee membership change events to occur at the same logical time on each member, events corresponding to sending and delivery of regular messages have to be ordered as well.

Formally, a process belongs to a group view in a virtual synchrony model if its history is *complete* and *legal*. An event history H is complete if:

C-1. All events which causally precede an event e, which belongs to H, also belong to H.

C-2. For each event $send(m)$ or $multicast(m) \in H$, there is an event $deliver(m) \in H$.

C-3. Each multicast message *m* delivered by a process P within view g^x is delivered by all other members of g^x.

While an event history H is legal if:

L-1. Each event *e* in H can be labeled with a global time *time(e)* that respects the causal order of events.

L-2. Each pair of events of the same process have distinct times.

L-3. Any two membership change events that install the same group view raised at two different processes occur at the same logical time.

L-4. The deliver events corresponding to a single multicast are all delivered in the same view of the group.

L-5. *Atomic delivery* is totally ordered. The delivery of an atomic message occurs at the same logical time at all delivering processes.

Note that conditions C-2 and C-3 require that if some failure or member join produces a membership change after a message is sent or multicast, but before the same message is delivered by at least one of the members, the delivery has to occur in the new group view obtained as a result of applying the membership change to the group view present when the message was sent. Similarly, a received message can not be delivered until the receiving member is sure that all other members of the current group view are able to deliver it.

Besides this, the virtual synchrony model requires a primary partition model (i.e., group partitions are not allowed) where faulty processes are eventually removed from the group and no additional message is delivered if it was sent by a faulty node. This forces recovered processes to change its identifier to be able to join the group again.

2.5.5 Extended Virtual Synchrony

Extended virtual synchrony [23] is the extension of the virtual synchrony model for membership protocols that tolerate partitions of the group view in several components and re-merging of these components at a later time. It preserves the same main properties as the virtual synchrony model, but in a harder environment. Thus, a given event may be ordered by any group member at the same logical time.

The *configuration* concept is introduced in [23] to refer to a group view together with a unique identifier. Two kinds of configurations are distinguished. In a *regular configuration* messages may be sent and multicast, and each member of the group delivers them following a given order. A *transitional configuration* is needed when a membership change has been found and some messages sent or broadcast in the previous regular configuration are still not delivered. No message can be sent in a configuration of this kind. This transitional configuration is used to deliver the messages of the old regular configuration, trying to respect the delivery semantics guaranteed by the type of order required when they were sent and having a group view as similar as possible to the previous regular

one. For instance, if a new member is added to the group, its addition forces the generation of a transitional configuration where it is not included yet. The new member will effectively join the group in the following regular configuration.

The delivery consistencies allowed in the original system where extended virtual synchrony was introduced are *causal delivery*, *agreed delivery* (total order) and *safe delivery* (delivery in a site implies delivery in all sites, and following total order.)

The events ordered in an extended virtual synchrony model are the delivery of a configuration change message, the sending of a regular message in a given configuration, the delivery of a regular message in a configuration and the failure of a given process.

The following properties define the extended virtual synchrony model:

1. *Basic delivery*. Events are totally ordered in a single process. Delivery of a message is preceded by its sending, and occurs in the same configuration or in the immediately following one. A message is sent only in a configuration and has only a sender process.

2. *Delivery of configuration changes*. If a process fails, the others detect the failure and install a new configuration. At any moment, a process only belongs to one configuration. Processes agree on the first and last message delivered in a configuration.

3. *Self-delivery*. A process delivers all messages it has sent, unless it fails.

4. *Failure atomicity*. If any two processes transit from a configuration to the next, then both processes deliver the same messages in that configuration.

5. *Causal delivery*. If a message is sent before another in a (regular) configuration and a process delivers the second one, it also has delivered the first one.

6. *Agreed delivery*. Agreed delivery is consistent with causal delivery, but extends it to consider also transitional configurations. It guarantees a total order of message delivery in each subgroup and allows to deliver a message as soon as all its predecessors in the total order have been delivered. In a transitional configuration there is no obligation to deliver messages sent by processes not in that configuration or in its previous regular configuration.

7. *Safe delivery*. If a process delivers a message in a configuration, then each process in that configuration delivers it unless that process fails. If any process delivers a safe message in a regular configuration, then all processes in this configuration had delivered a message to install this configuration.

Note that the causal, agreed and safe delivery orders are accumulative, i.e., agreed delivery implies also all the conditions required in causal delivery, and safe delivery requires all the conditions given in agreed delivery.

Observe that the main extension to the virtual synchrony model is the addition of transitional configurations which allow the delivery of pending messages before installing the new regular configuration. For instance, if a virtual synchrony model is used and a process broadcasts a message m when the current group view is {A, B, C} and before delivering it, a new process D joins the group, the message is delivered to D, too. In the extended virtual synchrony model, once the node D has been detected, a transitional configuration is formed without D, and the message m is delivered in it, preventing D from delivering that message. Another variation is that recovered processes can use the same identifier they had in its last configuration to rejoin the membership group.

The *strong* and *weak virtual synchrony* models [14] are similar to the extended virtual synchrony. They modify the virtual synchrony model to ensure that messages are delivered in the same configuration where they were sent. Furthermore, these models also cope with partitions and rejoins of subgroups.

2.6 Member Identification

Since a member process or site may leave the group either voluntarily or due to a failure and rejoin the group at a later time, some care must be taken to identify the potential and actual members of a group. The identifier assigned to a member may help or complicate the task of distinguishing whether messages sent by this member are valid or stale.

A member identifier is the name or code used by the membership protocol to identify an element of the membership set. This identifier is the one used to report changes in the group view to the client application. Each membership protocol may add other fields to this identifier. For instance, it may add a view sequence number to distinguish current group view members' messages from older ones. However, these additional fields are transparent to the client application and are never reported to it; thus, they do not belong to the member identifier.

The most common alternatives to identify group members are:

- *Invariable identifiers*. Each member of the group has an identifier that never changes. If the member fails and later recovers or the group view changes the member identifier is not modified.

  This is the approach followed by the Transis membership protocol [11], which uses the extended virtual synchrony model. As invariable identifiers are used, that protocol needs to insert a *context* field in each membership message. When the current configuration changes, its context number is increased. Detection of stale messages is achieved adding another field, a message counter, that is reset when the context number is increased.

  A similar approach is taken in [25] where the identifier never changes but an *incarnation number* is added each time a group member recovers. It presents several joining algorithms which assign incarnation numbers to new machines, either using stable storage to hold the last member incarnation number or not.

- *Identifiers per incarnation*. Each time a process or site member fails and recovers a new and unique identifier is assigned to it. This is the approach taken by the Isis membership protocol [27], which uses the virtual synchrony model.

  In this case, the identifier can be built using a per process invariable identifier which is expanded with an incarnation number. The main difference with invariable identifiers is that the client application finds new and unique identifiers when the members are recovered, and it can not distinguish between new members and faulty members being rejoined.

Member identification is also related to *failure stability* [28]. A membership service offers failure stability if once a group member is believed faulty its messages are rejected thereafter and, consequently, the group member is not allowed to recover. A service with failure stability uses *identifiers per incarnation* to permit member recoveries under a different identifier.

2.7 Partition Handling

A *partition* occurs in a communication group when at least two non-null and disjoint subsets of the group remain isolated; i.e., members in each one of the subsets are unable to communicate with members of any other subset. Conversely, a communication group is *partition-free* if any two operational nodes of the group can always communicate.

Three models of membership protocols are distinguished according to their partition admittance:

- *Absence of partitions model* [10]. This model assumes that partitions never happen in a communication group. To guarantee that hypothesis, enough redundant physical links among processors are assumed. Also, the intercommunication network may route messages through intermediate sites to bypass broken links.

  This model is followed in synchronous systems like the one described in [10] and [29].

- *Primary partition model* [28]. This approach recognizes that partitions may occur, but in that case only one of the subgroups is allowed to proceed. Members included in other subgroups are forced to fail and to retry their inclusion in the main group, as other new members must do. These protocols avoid the problem of re-merging subgroups.

  Group members need to know the maximum group size to be able to implement this model. Once this figure is known, the main group or the subgroups spawned in a partition can compare their size to the maximum one. Protocols in this model require a minimum part of the possible group of members to remain strongly connected, usually more than a half of the possible whole size. If the size of one of the current subgroups does not reach this minimum, the subgroup is forced to fail.

  This model is used in Isis [8, 27].

- *Partitionable system model* [13]. This model admits partitions and allows any resulting subgroup to proceed and re-merge at a later time. The problem of application state consistency must be solved when members belonging to previously isolated subgroups are merged.

  Systems with a partitionable model are Totem [1, 24], Relacs [2, 3], Transis [11], Phoenix [20] and Horus [14, 31].

The primary partition and partitionable models have several details that must be solved. For instance, how to force the failure of members of minor subgroups in the primary partition model and how are subgroups rejoined in the partitionable model.

Birman proves formally in [6] that several types of applications can not be run consistently in partitionable environments and must be executed in an absence of partitions or a primary partition model. His paper introduces the concept of *non commuting* actions and shows the impossibility of reaching consistency when a partitioned system tries to re-merge and rebuild its execution history if the events to be merged are non commuting.

2.8 Logical Interconnecting Network Topology

Messages are sent by a membership monitor through the network for different purposes. First, a membership monitor sends *periodical messages* to announce that it is still alive. When a member fails or a group member explicitly leaves the group, at least one monitor eventually detects this situation and tries to broadcast a message notifying that one member has left the group; this is a *leave message*. Finally, when an external process tries to join the group it has to send one or more *join messages* to at least one member of the group and, once it is accepted, some protocols require a *state transference message* to allow the joining process to know the state and identity of the current members.

Some membership protocols assume some special network topology, provided by the communication layer, and only transmit messages using the allowed channels. Others use different communication channels depending on the type of message being sent. For instance, periodical messages are sent through a logical ring and other message types are transferred assuming a fully connected network.

The most common interconnecting network topologies are:

- *Fully connected*. In a fully connected network each machine may send messages directly to any other machine in the system, since each pair of machines has a communication link. Even if the physical interconnecting network is not fully connected, a logical network of this kind can be built if some routing technique is used in the communications layer.

  This logical network topology has been assumed in Relacs [3], the *periodic broadcast membership protocol* of [10], Transis [11], Amoeba [17], TTP [19], Phoenix [20], Psync [22], Isis [27], Horus [31], DELTA-4 [29] and others [25]. In these protocols all types of messages are sent to any group member and assuming those processes directly accessible.

- *Ring*. In a logical ring each member process has two neighbors and only communicates with these two processes. In *unidirectional rings* each process receives messages from one of its neighbors and sends its messages to the other. In *bidirectional rings* messages may be sent to or received from any of them.

  The use of a logical ring is sometimes restricted to periodical messages only, using broadcasts in a logical fully connected network to transfer other messages. This approach is used in the *attendance list protocol* of Cristian [10], where the ring is unidirectional, and in its *neighbor surveillance protocol* and in [26], where the ring is assumed now bidirectional and each member sends periodical messages to both its neighbors.

  In Totem [1, 18] a unidirectional ring is assumed and the token being rotated through the ring is used to order broadcast messages as well as for detecting failures.

2.9 Symmetry

A membership protocol is *symmetric* (or fully distributed) if all membership monitors execute the same algorithm. A membership protocol is *centralized* if an special group member manages the behavior of the whole group, coordinating the acceptance of any group change.

A centralized membership protocol usually reduces the number of interchanged messages since explicit departures and joins may be forwarded to the manager member which accepts and makes them public. The same occurs with failures. Once they are detected, the manager is reported and the change is subsequently broadcast. However, the centralized protocol introduces a bottleneck in its manager member since all messages are sent to it or originated from it. Additionally, problems may arise if the manager is suspected faulty by other members because temporarily two or more managers coexist in the same group. The failure of the manager is not so problematic if the protocol establishes a criterion to elect its successor among the set of remaining live nodes.

The centralized approach has been followed by Isis [27], the strong, weak and hybrid protocols of [26] and the communication protocol of Amoeba [17]. Other protocols follow the symmetric approach, avoiding the problem of manager election when the current one fails but using a greater amount of messages when agreement must be achieved. An intermediate solution is achieved in [29] where special *changer* tasks run in each member. When a membership change has to be made, one of the members activates its changer task which tries to get a global lock. If the lock is obtained, this process becomes the current manager and initiates the necessary steps to change the current group view. Once all the members agree on the new changes, the lock is released and all members initiate their *guardian* tasks (the ones which exchange periodical messages) and the protocol behaves symmetrically again.

2.10 Agreement on Membership Changes

When the membership set changes, usually all membership monitors do not detect this change at the same time. Agreement is achieved when all the monitors in a set have detected the same changes and have updated its membership set accordingly. So, when agreement holds, all monitors consider the same membership set.

Two kinds of agreement exist:

- **Strong or regular agreement**. A membership protocol maintains regular agreement if all operational members see and have seen the same sequence of group views since they were included in the set. Observe that this kind of agreement forces membership change messages to be totally ordered.
- **Eventual agreement**. If eventual agreement is used, the protocol requires that when some membership changes occur, eventually all operational members will agree on the current membership set. This do not prevent two different members to have distinct group view sequences. For instance, an operational member A may have detected the failure and rejoin of a process C, while for member B the process C never failed.

Usually, regular agreement is used. However, the weak membership protocol of [26] uses eventual agreement. Eventual agreement is acceptable in some distributed applications, Rajkumar cites a tool to monitor a distributed system as a good example of an application of this kind, since the behavior of this application does not depend critically on the current state of all the nodes that build the system.

Another property related to agreement is how many members must agree initially to accept a membership change. The members that constitute this initial quorum set have to diffuse later their decision and all other members must accept it. The usual alternatives to build this initial quorum set are:

- **One member.** When a member suspects another is faulty or receives a new processor's request to join, it sends the message change to all other members, which accept the change and modify its copy of the membership set accordingly.

  This alternative is only used in asymmetrical protocols where a distinguished member can be found. In this case, the failure detection only reports to the leader, and this member broadcasts the changes to the regular members.

- **Majority subset.** The greater part of the current membership set agrees on the detected membership change.

  The Isis membership protocol [27] uses this kind of agreement on changes. In Isis a distinguished member (the manager) exists; it announces all changes to the other members. When a join or failure is detected by the manager, it sends a `submit` message reporting the change. All members which receive this message reply an `ack` message. When the manager has received the acknowledgments from a majority of the current members, it knows that agreement has been reached and multicasts a `commit` message. When the commit is received, the membership set is updated.

  If a majority subset of the current membership set does not respond with an `ack` message in a given time, the manager suspects that it has failed and leaves the group.

- **All members (Full agreement).** If a member is suspected faulty, all remaining members must know about the failure and agree on it. If a new member is detected, all current members must receive some message about this fact before all of them agree on its joining and the new member is added to the membership set.

  This behavior can be found in the Psync protocol [22]. In this protocol all membership-related messages are sent to all current members. If a message is sent after receiving another one, the second logically precedes the first one. Two messages have the *same logical time* if neither of them precedes the other.

  In Psync there is an external detection protocol which sends messages about the possible failure of one or more current members. When a message of this kind arrives, all members check if some message from the member suspected faulty has arrived at the same logical time. If so, a negative acknowledgment is replied. Otherwise, a positive acknowledgment is sent. If some reply is a negative acknowledgment, the member suspected faulty is maintained in the membership set. To delete a member from the set, all current members must agree and send a positive acknowledgment.

  Similarly, to accept a new member the detection protocol introduces a message announcing the new member. When a member detects that all members have acknowledged the announcing message, it inserts the new member in the membership set. Note that all membership changes require agreement among all current members, but each member can detect locally when this agreement is achieved. Since all members run the same program and do not need a leader to manage this agreement phase, the Psync protocol is symmetrical.

Although all membership protocols need majority or full agreement, they reach it using different procedures. The agreement protocols described above are only samples. For instance, the two phase commit protocol used by Isis to reach majority agreement is also used by the strong membership protocol of [26]. Transis [11] uses full agreement, but its algorithm is quite different from the 2PC of Isis or the one described in Psync.

Another property related to agreement is safety, or how to force that membership changes taken using the agreement property are respected by the members excluded from the group. To guarantee that the decision taken about a membership change is safe, in addition to the agreement protocol we need a mechanism to discard stale messages sent by processors considered faulty. Usually, this mechanism is implemented adding the group identifier (or group sequence number) to each message sent by any member of the group. The group identifier is increased each time a membership change arises and the group contents are modified. So, faulty members can not communicate with the group if they do not rejoin it, since they do not know the current group identifier. Stale messages are easily detected with this mechanism because they have an obsolete group identifier.

2.11 Failure Detection

Although some membership protocols consider the failure detection mechanism as an external service, it is very important because it determines the accuracy and liveness of the resulting membership protocol and may restrict the assumed failure model. As it has been stated in a previous section, the ideal failure detection mechanism is accurate and live, but this two properties can not be achieved in asynchronous environments. In [30], the lack of accuracy is given as a reason to prevent the use of a fail-stop failure model, since using an inaccurate failure detector, the protocol can not be sure about the failure of a given member.

The greater part of the membership protocols seem to assume a fail-stop failure model. According to [30], this failure model is based in two conditions:

1. If one processor fails (crashes), all other processors eventually detect the failure of this node or fail, too.
2. There are no false detections of failure; i.e., when a processor is reported faulty, it actually has failed.

In asynchronous systems, the second condition is difficult to achieve. So, many asynchronous protocols simulate a fail-stop failure model. To simulate this failure model, the common solution is to relax the second condition stated above. So, in simulated fail-stop models there may be false detections of failure, partially corrected forcing the suspected faulty node to fail and all the remaining nodes to agree on this failure.

Besides the failure model, the failure detection mechanism has to be implemented, too. The most common implementations are:

- Member processors exchange periodical “I am alive” messages and the failure detection mechanism is based in timeouts. When some periodical messages are lost from a given processor, this processor is suspected

faulty. If the membership protocol reaches agreement on this failure, the processor is deleted from the current membership set.

- There is an external communication service which reports to the membership service any failure suspicion.

Note that the first alternative is implemented by the membership algorithm, while the second one relies on external services. When an external communication service provides the failure detection, the implementation of the failure detection depends on the services and warranties provided by this communication layer. For instance, if this service is based on acknowledged messages, it may report a failure when no acknowledgment has been received when the service tries to send a message to some particular node, and this attempt has been repeated a given number of times. Anyway, the membership protocol is independent in that case of the implementation of the failure detection mechanism. The communication layer may change this implementation without forcing any modification of the membership protocol.

2.12 Startup and Recovery Procedures

When the members of a group start, two approaches may be taken to build the membership set. The first is to know in advance which are the preconfigured members of the group. The second approach has no knowledge about the possible members. According to [15] these startup alternatives receive the following names:

- **Collective startup**. In this case, all members know at startup time the identifiers and addresses of the other possible group members. So, in this procedure the joining member is able to send a message to all other preconfigured members of its group. In this moment a new membership set may have been built and the joining node may know the new membership list following the contents of the answers received.

  This startup procedure will work either for the initial global startup — when the first membership group is built — or for the case of a node join. To work in both cases, the contents of the messages exchanged have to be carefully designed. As a minimum, the identifier of the joining node has to be in the startup or join request message and the membership set has to be present in the answer message.

- **Individual startup**. If individual startup is used, each member does not know in advance the identity or address of its companions. This startup procedure is more complex than the collective one. Initially, each member has to build a singleton membership set where only itself is included. The following step consists in trying to exchange some message with other isolated members or with some member of the current group membership set. This requires to broadcast an initial message. If somebody answers this broadcast, the membership sets of the sender and the replier have to be merged, anybody has to know the physical addresses of all other members and logical identifiers have to assigned to the new members.

  Individual startup offers the advantage of its flexibility, since it does not require to know the identity of all group members in advance. So, if this startup procedure is used, it is easier to add new nodes to the group. Note that collective startup requires that the new nodes were included in some configuration file or in some server which provides the group set to the starting members.

A *recovery procedure* deals with restarting a faulty member and making it to rejoin the group. There are some elements which determine the recovery procedure to be used. The first one is the startup procedure used by the protocol. A recovery is very similar to a startup, so the join of the recovered member will use the same steps needed to start a new member. Therefore, if the recovered node knows the identity of the other members it will use a collective startup procedure to rejoin, otherwise an individual startup procedure is used.

Member identification may influence the recovery procedure. Some protocols, as those which use virtual synchrony, require different identifiers per incarnation. In these protocols the recovery procedure has to deal with assigning a new identifier to the recovered member. Also, this identification approach complicates the use of a collective startup procedure to rejoin the new member since the set of member identifiers is not invariable and the recovering member can not know the current member identifiers.

Some protocols also require special message delivering orders and guarantees. In some cases, a protocol demands that all group members have delivered the same messages and in the same order. To accomplish this requirement, recovered members must receive and deliver a copy of the messages lost when the member was out of order, and these messages must be transferred in the recovery procedure. In other protocols, the recovered

member must discard all messages sent in previous group configurations. In those cases, the recovery procedure has no additional task to do.

As we have explained, the recovery procedure depends on the level of guarantees provided by the membership protocol. If a protocol uses individual startup, different identifiers per incarnation and a strict message delivery order, its recovery protocol will be complex. On the other hand, using a collective startup procedure, invariable member identifiers and no message delivery order, the recovery protocol will be as simple as its startup one.

3 Example Protocols

Table 1 summarizes the characteristics offered by some membership protocols described below, where some details of the algorithms used by each protocol are outlined. The properties given in this table and their abbreviations are the following:

- **Synchronism (Sy)**. The target system may be synchronous (S) or asynchronous (A).
- **Accuracy (Ac) and Liveness (Li)**. These two columns report if the protocols accomplish any of these properties.

	Sy	Ac	Li	Fi	To	Af	Al	Vs	Ev	Id	Pa	Ne	Sm	Ag	Nu	Fd	St
Cristian 1	S	*	*	*	*					U	A	F	S	S	A	H	I
Cristian 2	S	*	*	*	*					U	A	R	M	S	A	H	I
Cristian 3	S	*	*	*	*					U	A	R	S	S	A	H	I
Delta-4	S	*	*	*	*					U	A	F	M	S	1	A	C
Isis	A		*	*	*	*	*	*		I	1	F	M	S	M	E	C
Phoenix	A		*	*	*	*	*	*		U	1	F	M	S	A	E	
Psync	A		*	*	*	*	*			U	1	F	S	S	A	E	
Strong	A		*	*	*					U	P	R	M	S	A	H	I
Totem	A		*	*	*	*	*	*	*	U	P	R	S	S	A	T	I
Transis	A		*	*	*	*	*	*	*	U	P	F	S	S	A	E	I
TTP	S		*	*	*	*	*			U	A	F	S	S	M	A	C
Weak	A		*							U	P	R	M	E	1	H	I

Table 1: Main characteristics of some membership protocols.

- **Event ordering**. There are several event orders that can be supported by a membership protocol: FIFO order (Fi), total order of membership changes (To), agreement on first message (Af), agreement on last message (Al), virtual synchrony (Vs) and extended virtual synchrony (Ev).
- **Member identification (Id)**. A membership protocol may use invariable identifiers (U) or identifiers per incarnation (I) to refer to its members.
- **Partition support (Pa)**. A membership protocol has three alternatives to handle partitions: to assume that partitions will never occur, i.e., absence of partitions (A); let only the primary partition to survive (1); and let all partitions to coexist (P).
- **Logical network topology (Ne)**. It can be a fully connected network (F) or a logical ring (R).
- **Symmetry (Sm)**. The algorithm may be symmetrical (all members use the same algorithm and play identical roles) (S) or with a distinguished member which manages all membership changes (M).
- **Agreement**. Agreement (Ag) may be strong (S) or eventual (E). Another classification can be made according to the number of members (Nu) needed to reach agreement: only one (1), a majority (M) or all the members (A).
- **Failure detection (Fd)**. It can be based on heartbeats or periodical messages (H), requiring explicit acknowledgments to all sent messages (A), token loss in a logical ring (T) or using external services (E), as a dedicated failure detection module or a communication server.

- **Startup procedure (St)**. There are two types: individual startup (I) and collective startup (C).

In the following paragraphs more details about each protocol can be found. Note that only a brief description of some algorithms is given. The interested reader may find more information using the referenced bibliography.

3.1 Cristian's Membership Protocols

Cristian [10] describes three algorithms for synchronous environments. All the algorithms use broadcasts to join new members to the group. The applying process issues a `NEW_GROUP` broadcast which is subsequently replied with a `PRESENT` broadcast for each current member of the group. Once the time bound to get `PRESENT` replies is reached, all current members agree on the new membership set. The same procedure is used when some member failure is detected and a new membership set has to be computed. In that case, the failure detector initiates the protocol.

Besides this join procedure, all algorithms share other properties and services, such as atomic broadcasts or the absence of partitions assumption. The main differences come from the mechanism used to detect member failures. Each protocol uses different mechanisms, needing fewer messages the last algorithms:

1. **Periodic broadcast membership protocol**. To check the stability of the group, each member broadcasts periodically a `PRESENT` message. Since all members are synchronized, the absence of some message means that its sender has failed.

2. **Attendance list membership protocol**. The current members are arranged in a logical ring and a distinguished member is chosen (following a decreasing member identifiers order). Periodically, this leader sends the membership list to its successor in the order, which transmits it to its successor and so on. If the list does not arrive in time, the member which expected it suspects the failure of its predecessor and reports it to the group.

3. **Neighbor surveillance protocol**. As in the previous algorithm, all members are arranged in a logical ring. Periodically, each member sends a message to one of its neighbors (the one with lowest identifier, for instance) and expects the same message from its other neighbor. If the message does not arrive, this neighbor is assumed faulty.

3.2 Delta-4

The membership protocol used in the Delta-4 [29] real-time system is also synchronous and bases its failure detection mechanism on a transmission with response procedure. The greater part of the messages broadcast in this protocol require an acknowledgment. If some member does not send this acknowledgment to the leader, it is assumed faulty.

Another characteristic of the protocol is how a member becomes the leader. Usually, no distinguished member is present. When some member detects a problem to communicate with another one it tries to become the current leader, modify the membership set, diffuse it and return to its regular state. To be elected as a leader, a member has to broadcast a message announcing its intention. Then it waits for the acknowledgments of the other members. If all members reply affirmatively, this member is the new leader and is able to modify the membership set (either to add a joining member or to take out a faulty one). Note that if two or more members try to be elected as leaders at a time, a deadlock can occur. The protocol solves this situation using a message count field which is increased each time a collision among several applying members is detected.

3.3 Isis

Isis [8] is a group-oriented communication environment for asynchronous distributed systems. Its Strong GMP (group membership protocol) [27] uses external failure detectors which report all membership change events to a distinguished member, the manager. This manager leads regular members through the protocol phases.

When its group view changes, the manager broadcasts the membership change in a `SUBMIT` message, which has to be answered by the regular members with an `ACK` message. Once the manager has collected a majority of acknowledgments, it broadcasts a `COMMIT` message and assumes the new group view has been accepted and

installed. If the manager is unable to receive a majority of acknowledgments it assumes itself faulty or isolated in a minor partition. Its response to this situation is to crash itself; i.e., it stops sending and receiving messages.

When the membership change corresponds to a member join, the manager sends a STATE-XFER message to the new member in the commit phase. In this message, the new process receives permission to join and some information about the current group state. Thus, the new member knows that it has been included in the group.

Although the use of a manager reduces the number of needed messages, also complicates the protocol in case of a manager failure. Then, a new manager must be chosen among the remaining members and an additional phase is needed to reach agreement on the membership change. This new phase is initiated before the submit one and is needed to know the state of each member when the manager crashed.

Isis has been evolved to Horus [31], which tries to offer the same properties of its predecessor but tolerating partitions and improving its performance. The architecture of Horus is based on stackable micro-protocols. Each micro-protocol offers a different property or service. For instance, the type of event ordering used, secure extensions, hierarchical membership services, etc. With Horus, different protocols can be built using different subsets of all the available micro-protocols. The same idea has been used in other modern group communication toolkits, such as Consul [16].

3.4 Phoenix

Phoenix [20, 21] is an asynchronous group communication protocol similar to Isis but oriented to large scale networks. Its membership subprotocol maintains the Isis primary partition model but uses an *unstable suspicion model* to maintain this primary partition. As a result, this protocol uses invariable identifiers to refer to its members, since a failure detector may realize that a member previously considered faulty is now up and running. So, once a failure is detected, the faulty node is not forced to crash for ever as is done in Isis.

This protocol distinguishes three kinds of group members: core members, client members and sink members, as explained in section 2.4. The support and services provided to members vary according to their kind. Core members, for instance, are the only class which has a guaranteed message delivery order.

While Isis provides the virtual synchrony model to deal with message delivery and membership changes, Phoenix uses the *view synchronous communication* model which improves the former in some minor details.

3.5 Psync

Psync [22] uses an external server which reports the membership protocol about failures or joins. So, the Psync protocol deals only with checking whether the membership change reported is true or not and with reaching agreement about the change to be done.

To accept a new member the external detector sends a message announcing the start of that process. The current members reply to this start message broadcasting an acknowledgment. Then, if each member has collected an acknowledgment from all the current members, the new one is included in the set.

To take out a faulty member the same procedure is needed. Unlike the join case, now the members may broadcast a NAK message if they have received a message from the suspected faulty member which can not causally precede the received failure notification. Since both messages have been sent at the same logical time, the failure notification is rejected. So, if one of the members replies with a NAK message the member is not considered faulty and is not taken out of the membership set. As we can see, this protocol requires full agreement to accept a membership change event.

3.6 Strong and Weak Protocols

The strong and weak membership protocols [26] assume an asynchronous distributed system and are presented as two alternatives to solve the membership problem. The weak membership protocol is useful for applications which do not require an immediate group change notification. It uses eventual agreement; i.e., if a membership change occurs, different members react to the change at different speeds and some of them even do not detect it, but eventually all members will agree on the resulting membership set. The strong membership protocol requires that all group members have the same history of events. Therefore, all running members must have seen from their startup the same sequence of group views and in the same order.

In both protocols the current group is structured in a logical ring, and a member is chosen as the group leader. To deal with failure detection, each member sends periodically a message to each of its two neighbors in the ring.

When a failure has been detected in the weak protocol, the detector reports the change to the leader. Then, the leader broadcasts a NEW_GROUP message which is accepted by all other members. Regular members choose their two neighbors from the components of the new group and start to send the heartbeats until new changes are detected.

In the strong protocol this algorithm is extended with a second phase. When the leader receives the failure notification, it broadcasts a PTC (Prepare to commit) message. The regular members reply with an ACK or NAK message. If no NAK has been received, the new membership set is broadcast by the leader in a second COMMIT message and accepted by all members.

Since both protocols are centralized, both of them have to deal with the failure of the leader. In that case, another member (the one with the highest identifier) is chosen and it has to manage the next steps of the protocol.

3.7 Time-Triggered Protocol

The Time-Triggered Protocol [19], or TTP for short, consists of a series of services for a real-time distributed system. It includes a predictable message transmission service, a clock synchronization service and a membership service among others. Communication services rely on TDMA rounds, each one for a different (and possibly replicated) node (or fault-tolerant unit, following the TTP naming). Since clock synchronization services are used, the membership protocol assumes a synchronous environment.

Each fault-tolerant unit must send at least a message in each of its TDMA rounds. When a node detects some communication problems (it does not receive any acknowledgment or receives all the frames in a corrupted state) stops and remain inactive. Thus, when some members detect that another one is not using its TDMA round, they mark this node as faulty and in the following TDMA round it is removed from the membership set.

3.8 Transis and Totem

Transis [11] provides a group communication environment for asynchronous systems. Compared to Isis, it offers support for partitioned group operation, partition re-merging and a stronger event ordering model: the extended virtual synchrony. Another important difference with Isis is that this protocol does not need a manager; i.e., the Transis protocol is symmetrical.

The resulting membership protocol deals also with message delivery order and support for partitioned operation. So, it is quite complex and is not described here.

Totem [1, 18, 24] is a variation of the Transis protocol which provides the same services but uses a logical token-passing ring to decrease the number of messages needed by the protocol.

4 Additional Work

The group membership protocols described in previous sections cover a broad spectrum: synchronous and asynchronous environments, accurate and live failure detectors, from no event ordering (as in the Weak protocol) to very strong event orderings (e.g., the extended virtual synchrony), from no partition handling to support for partitioned operation and state consistency on partition re-merging, from symmetrical algorithms to algorithms managed by only a distinguished member, from full agreement to decissions taken by only one member, etc. But there are still some areas where nowadays the appropriate protocol can not be found.

Sometimes we have a group communication toolkit which uses only a membership protocol. This membership protocol has to offer enough services to all its client applications. So, when there are some alternatives for a given protocol property (e.g., event ordering model, startup procedure, partition handling), the protocol offers the strongest one to be sure that if some client requires this kind of service, it obtains it. In that case, all client applications that do not require this extra property have to pay for it. So, the ideal approach is to have a configurable membership service, such as the one described in [16], where an evolution of the Consul protocol is explained. Now, when some client applications need the membership services they initiate it configuring the kind of services needed. Therefore, the membership server is tailored to the needs of its clients and the latter do not have to pay for services never used. Protocols of this class can be found in the last releases of Consul and Horus. Both approaches use a big set of stackable micro-protocols which can be combined in different ways to obtain the appropriate protocol. However, the solution given in Consul seems to have a finer granularity than the one presented in Horus; i.e., the Consul micro-protocols permit to choose among individual properties (Horus only have a reduced set of

micro-protocols related to membership, the others deal with other communication features) and to build any kind of membership protocol.

Another interesting area are large scale networks which require group membership protocols capable to deal with very large group views. Some work has been started in this area and the first protocols are Phoenix and Horus. Phoenix uses a dedicated protocol developed from scratch. Horus adds some micro-protocols to allow hierarchical membership sets and to enlarge in this way the capabilities of the managed group views.

One of the latest trends in group membership protocols has been the support of partitioned operation, which was introduced by the Transis protocols and its predecessors. The greatest problem in this area is how to achieve a consistent state of the client application once the partitions have been re-merged. There are no obvious solutions to this problem and usually the membership protocol does not help too much. The client application has to deal with the task of event re-ordering if it wants to merge the updates made in each isolated partition. The easiest approach is to forget the updates made in minor partitions and to impose the state of the primary one, but for some applications this solution is not allowed. Some support could be offered if the membership and communication services keep track of part of the messages received by the isolated partitions. The viability of this support deserves further study.

A possible extension to the group membership services could be the maintenance of the set of running communication links among the current members. Thus, the communication layer could check this information and re-route the messages avoiding broken links.

As we have seen, group membership services is not a static research area. Although the basic algorithms have been thoroughly described in the last years, new extensions may be identified and studied.

5 Conclusions

This paper describes the group membership service and the systems which need this kind of service. Implementations of a group membership service vary from synchronous to asynchronous distributed systems. In the first case, the membership protocol may satisfy the accuracy and liveness properties while in asynchronous systems one of them can not be accomplished, and liveness is usually the property maintained. Other properties have been analyzed and various alternatives presented for each property. Thus, a membership protocol may be characterized by its event ordering model, its degree of centralization, its support for partitioned operation, the kind of logical network used for message exchange, its failure detection mechanism, how agreement is achieved, its startup procedure, etc.

Current trends in group membership services are the extension of traditional membership services to support partitioned operation, partition re-merging and stronger event ordering models. This alternative is followed by Totem and Transis. They try to offer a membership service who is able to run in environments where other membership services can not, and incurring in similar communication costs.

Another trend is the use of a collection of stackable micro-protocols, which can be combined to form a group communication protocol. The service provider may choose the most appropriate subset of micro-protocols for the quality of service required in the system. So, if some applications do not require a strong event order model, they do not have to pay for it. The resulting protocol does not include any service that the application or system does not need. As a result, performance is better and the service provided can be easily adapted to the target system. The last releases of Horus and Consul use this approach.

Finally, group membership services have to be shown as an important component of current systems. Although they were initially needed only by some replicated applications or by group communicating environments, nowadays distributed systems have increasing importance and all distributed applications need some type of membership service to maintain a consistent shared state.

References

[1] Y. Amir, L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, and P. Ciarfella. Fast message ordering and membership using a logical token-passing ring. In *Proceedings of the 13th International IEEE Conference on Distributed Computing Systems*, pages 551–560, May 1993.

[2] Ö. Babaoğlu, R. Davoli, L. A. Giachini, and M. Baker. Relacs: A communications infrastructure for constructing reliable applications in large-scale distributed systems. Technical report, UBLCS-94-15, Dept. of Computer Science, University of Bologna, Bologna, Italy, June 1994.

[3] Ö. Babaoğlu, R. Davoli, and A. Montresor. Failure detectors, group membership and view-synchronous communication in partitionable asynchronous systems. Technical report, UBLCS-95-18, Dept. of Computer Science, University of Bologna, Bologna, Italy, November 1995.

[4] Ö. Babaoğlu and A. Schiper. On group communication in large-scale distributed systems. Technical report, UBLCS-94-19, Dept. of Computer Science, University of Bologna, Bologna, Italy, July 1994.

[5] J. M. Bernab´eu, Y. A. Khalidi, V. Matena, K. Shirriff, and M. Thadani. The design of solaris mc: A prototype multi-computer operating system (2nd edition). Technical report, SMLI-94-492, Sun Microsystems Laboratories Inc., Mountain View, CA, June 1995.

[6] K. P. Birman and R. Friedman. Trading consistency for availability in distributed systems. Technical report, TR96-1579, Dept. of Computer Science, Cornell University, Ithaca, NY, April 1996.

[7] K. P. Birman and B. B. Glade. Consistent failure reporting in reliable communication systems. Technical report, TR93-1349, Dept. of Computer Science, Cornell University, Ithaca, NY, May 1993.

[8] K. P. Birman and R. van Renesse. *Reliable Distributed Computing with the Isis Toolkit*. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[9] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. Technical report, TR95-1548, Dept. of Computer Science, Cornell University, Ithaca, NY, October 1995.

[10] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 6(4):175–187, 1991.

[11] D. Dolev, D. Malki, and R. Strong. An asynchronous membership protocol that tolerates partitions. Technical report, CS94-6, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, March 1994.

[12] D. Dolev, D. Malki, and R. Strong. A framework for partitionable membership service. Technical report, CS95-4, Insitute of Computer Science, The Hebrew University of Jerusalem, Israel, 1995.

[13] R. Friedman, I. Keidar, D. Malki, K. P. Birman, and D. Dolev. Deciding in partitionable networks. Technical report, 95-16, Institute of Computer Science, The Hebrew University of Jerusalem, Israel, November 1995.

[14] R. Friedman and R. van Renesse. Strong and weak virtual synchrony in horus. Technical report, TR95-1537, Dept. of Computer Science, Cornell University, Ithaca, NY, August 1995.

[15] M. A. Hiltunen and R. D. Schlichting. Understanding membership. Technical report, 95-07, Dept. of Computer Science, The University of Arizona, Tucson, AZ, July 1995.

[16] M. A. Hiltunen and R. D. Schlichting. A confi gurable membership service. Technical report, 94-37A, Dept. of Computer Science, The University of Arizona, Tucson, AZ, January 1996.

[17] M. F. Kaashoek, A. S. Tanenbaum, and K. Verstoep. Group communication in amoeba and its applications. Technical report, Dept. of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, 1993.

[18] M. R. King, L. E. Moser, P. M. Melliar-Smith, and D. A. Agarwal. Monitoring membership changes in a fault-tolerant distributed system. In *Proceedings of the 8th International Conference on Parallel and Distributed Computing Systems, Orlando, FL*, pages 1–8, September 1995.

[19] H. Kopetz and G. Grünsteidl. Ttp - a protocol for fault-tolerant real-time systems. *IEEE Computer*, 27(1):14–23, January 1994.

[20] C. Malloth, P. Felber, A. Schiper, and U. Wilhelm. Phoenix: A toolkit for building fault-tolerant distributed applications in large scale. Technical report, D´ept. d'Informatique,École Polytechnique F´ed´erale de Lausanne, Lausanne, Switzerland, July 1995.

[21] C. Malloth and A. Schiper. View synchronous communication in large scale networks. Technical report, D´ept. d'Informatique,École Polytechnique F´ed´erale de Lausanne, Lausanne, Switzerland, 1995.

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

[23] L. E. Moser, Y. Amir, P. M. Melliar-Smith, and D. A. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems, Poznan, Poland*, pages 56–65, June 1994.

[24] L. E. Moser, P. M. Melliar-Smith, D. A. Agarwal, R. Budhia, and C. Lingley-Papadopoulos. Totem: A fault-tolerant multicast group communication system. *Communications of the ACM*, 39(4):54–63, April 1996.

[25] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):459–473, May 1994.

[26] R. Rajkumar, S. Fakhouri, and F. Jahanian. Processor group membership protocols: Specifi cation, design and implementation. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems, Princeton, NJ*, pages 2–11, October 1993.

[27] A. M. Ricciardi. The group membership problem in asynchronous systems. *Ph.D. dissertation (also available as TR92-1313), Dept. of Computer Science, Cornell University, Ithaca, NY*, page 198 pgs, January 1993.

[28] A. M. Ricciardi, A. Schiper, and K. P. Birman. Understanding partitions and the "no partition" assumption. Technical report, TR93-1355, Dept. of Computer Science, Cornell University, Ithaca, NY, June 1993.

[29] L. Rodrigues, P. Veríssimo, and J. Rufi no. A low-level processor group membership protocol for lans. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 541–50, May 1993.

[30] L. S. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. Technical report, TR94-1413, Dept. of Computer Science, Cornell University, Ithaca, NY, June 1994.

[31] R. van Renesse, K. P. Birman, B. Glade, K. Guo, M. Hayden, T. M. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical report, TR95-1500, Dept. of Computer Science, Cornell University, Ithaca, NY, March 1995.