A Configurable Membership Service

Matti A. Hiltunen Richard D. Schlichting

TR 94-37A

A Configurable Membership Service[1]

Matti A. Hiltunen Richard D. Schlichting

TR 94-37A

Abstract

A membership service is used to maintain information about which sites are functioning in a distributed system at any given time. Many such services have been defined, with each implementing a unique combination of properties that simplify the construction of higher levels of the system. Despite this wealth of possibilities, however, any given service only realizes one set of properties, which makes it difficult to tailor the service provided to the specific needs of the application. Here, a configurable membership service that addresses this problem is described. This service is based on decomposing membership into its constituent abstract properties, and then implementing these properties as separate software modules called micro-protocols that can be configured together to produce a customized membership service. A prototype C++ implementation of the membership service for a simulated distributed environment is also described.

December 19, 1994

Revised January 9, 1996

Department of Computer Science
The University of Arizona
Tucson, AZ 85721

[1]This work supported in part by the Office of Naval Research under grants N00014-91-J-1015, N00014-94-1-0015, and N00014-96-0207.

1 Introduction

Many types of highly dependable applications—that is, applications that are relied on to provide service despite failures [Lap92]—are implemented using distributed systems in which multiple machines are connected by a communication network. For such applications, the problem of keeping track of which machines are functioning and which have failed at any given time is vital. This problem is often called the *membership problem*. A distributed service that maintains consistent information at all sites about the membership of a group of machines or, equivalently, processes is called a *membership service*, while the algorithm or implementation that realizes the service in a distributed system is called a *membership protocol*. Both membership services and protocols have been the subject of a large number of papers in recent years [ADKM92a, AMMS+95, Cri91, DMS94, EL95, EL90, GT92, KGR91, LE90, MSMA94, MPS92, MAMSA94, RFJ93, RB91, Rei94, SM94, SR93, SCA94].

While this variety of membership services and protocols gives the user many options, in most cases each service provides only a single combination of properties optimized for a given situation. As a result, the system designer often has little choice and may end up having to use a protocol that is either too strong or too weak. For example, the membership service in the ISIS system implements *virtual synchrony* [BSS91], which is a relatively strong property that guarantees that messages reflecting membership change events are delivered to the application at every site at precisely the identical point in the message stream. While appropriate and useful for many applications, it can unnecessarily restrict concurrency between processes if used when a weaker guarantee would suffice. This type of mismatch has led to many different membership properties being defined, but in the context of separate services rather than as part of a single service presenting multiple options to the user.

In this paper, we describe a single highly configurable membership service in which properties and their corresponding execution guarantees can be tailored to the specific needs of the application. With this service, for example, the user can decide at configuration time whether to include a message-ordering guarantee such as virtual synchrony, and if so, what variant. Our approach is based on decomposing membership into its constituent abstract properties, and then implementing these properties as a collection of software modules called *micro-protocols*. A custom service is then constructed by selecting micro-protocols corresponding to the desired properties, and linking them together with a runtime system implementing an event-driven execution model. The result is a software subsystem that can be used in conjunction with other message-passing protocols to form a network layer for machines involved in the application. In a larger context, this work can be viewed as extending the hierarchical approach to constructing modular networking software represented by such systems as the *x*-kernel [HP91] and Horus [RBG+95, RB95] to support finer grain modules and more flexible inter-module communication capabilities.

The goal of this paper is to describe the design of this configurable membership service and to relate some initial experience with a prototype implementation written in C++. In relation to other papers on membership, our primary focus is on a new approach to designing and implementing such services, with the specific algorithms being a secondary contribution. Our design allows the choice of properties in a variety of areas. These areas include whether the service is accurate or live, what kind of agreement is performed for suspected failures and recoveries, how messages are ordered, and how partitions are handled. This work builds on earlier papers that describe the abstract properties of membership services [HS95b, HS95c], as well as related work in which this approach has been used for other services such as atomic multicast and group RPC [GBB+95, HS93, HS95a]. The membership service will eventually be ported to an *x*-kernel based system that is currently under development [BS95].

2 Preliminaries

2.1 System structure

A membership service can be viewed as an underlying software layer that generates messages indicating changes in membership and forwards them to higher levels. These *membership change messages* can report, for example, failures, recoveries, or the joining of two partitions. Membership can be characterized in terms of any entity in a distributed system for which current status information is required, such as processes in a process group, processors, or larger entities such as entire computing systems. Here, *sites* are assumed to be the entity of interest, so that membership change messages refer to such events as site failures and recoveries within a specified group of interest. We use the term *group member* to refer to an unspecified site within this group. The network is assumed to be *asynchronous* with no *a priori* time bounds on message delivery. The failures considered are site crash and performance failures, as well as typical network failures such as lost messages.

Given such a system, the properties of a membership service can be defined in terms of what membership change messages it generates and when they are delivered to the application [HS95b]. The key abstraction for defining and implementing such a service is an *ordering graph*, i.e., a graph in which the nodes are application and membership change messages, and the edges are ordering constraints between the messages. Specifically, the edges define the predecessors of a message, where the predecessors of a message M are those messages that must be delivered to the application before M can be delivered. Several collections of protocols for building fault-tolerant systems are based on

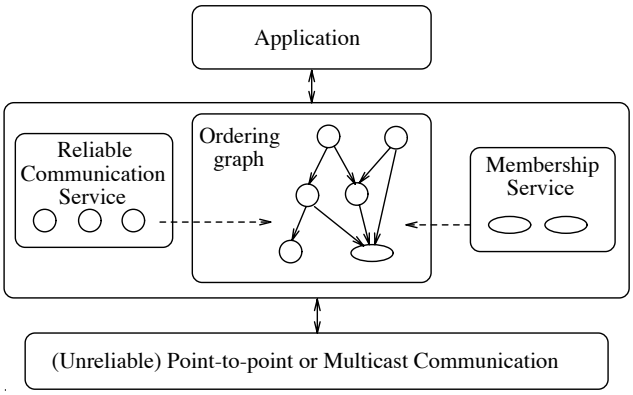

Figure 1: System structure

abstractions resembling ordering graphs, including Consul [MPS93a] and Transis [ADKM92b].

Figure 1 illustrates the logical system structure. The application, which is realized as a process group executing on multiple sites, is the top layer. The *reliable communication* and *membership* services add application and membership change messages, respectively, to the ordering graph. The reliable communication component is responsible for realizing reliable ordered multicast communication between group members; it guarantees that every message multicast by a group member will be received at all sites that remain functioning for the duration of the multicast. Since ordering properties like FIFO or causal ordering are defined relative to the sending site, these application messages typically encode in some way the information needed to realize these constraints. The responsibility of the membership service is to guarantee that membership change messages appear in the ordering graph when and where they are supposed to according to the properties specified.

2.2 Properties of membership

As noted in section 1, any given membership service implements some specified collection of properties. A number of these properties are defined formally in [HS95b, HS95c] and summarized below.

Accuracy and Liveness. Accuracy and liveness deal with detecting a change in status of a group member, either from functioning to non-functioning (*failure*), or from non-functioning to functioning (*recovery*). An *accurate* membership service is one that detects a change only if the change has indeed occurred (i.e., no false detections), while a *live* membership service is one that is guaranteed to detect all changes eventually [BG93]. In an asynchronous system, it is impossible to have a membership service that is both live and accurate [CT91, FLP85]. We can further distinguish between properties related to detecting failures and those related to detecting recoveries. For example, in most systems, failure detection is live, while recovery detection is accurate.

Confidence. The *confidence* property is the degree of certainty in a suspected membership change. The typical way to increase certainty is to collect information from multiple sites before making the final decision on whether to forward a membership change message to the application. Different variants of the confidence property can be realized by specifying different rules for this decision. Possibilities range from allowing the decision to be made by only a single site (*single site suspicion*) [RB91, RFJ93, SM94] to requiring that all functioning sites agree (*consensus*) [MPS92, AMMS+95]. The possibilities between these two extremes, called *voted decision* in this paper, are not commonly explored in existing systems, a notable exception being [Rei94].

Agreement. The *agreement* property states that if one site delivers a membership change message to the application, all other functioning sites will eventually deliver the same message. When a partition occurs, this property applies only to those sites within the same partition.[1] Weaker forms of agreement can also be identified. For example, *eventual agreement* guarantees that the membership views on different sites will eventually converge but the set of membership change messages received may be different [RFJ93].

Ordering. Ordering properties specify whether membership change messages are ordered consistently at all sites with respect to each other and/or application messages, and if so, what ordering is enforced. Examples of orderings involving only membership change messages include *FIFO order*, which implies that sites deliver membership change messages reporting the failure or recovery of a given site in the same order, and *total order*, which implies that sites deliver membership change messages involving *all* sites in the same order. Examples of orderings involving application messages include:

- *Agreement on last message*. A final message from the failed site is agreed upon and delivered at all sites before the change message for the failed site is delivered. An analogous property, *agreement on first message*, can be defined for recovery.
- *Agreement on predecessors*. A set of messages is agreed upon and delivered at all sites before delivering the change message. An analogous property, *agreement on successors*, defines an agreed set of messages to be delivered after the change message.
- *Virtual synchrony*. Membership change messages are delivered to the application at the same point in the application message stream at all sites.

[1] Sites in other partitions or that were failed during the agreement process will typically obtain updated membership information during the partition merge or recovery process, respectively.

- *Extended virtual synchrony*. An extension to virtual synchrony where all messages sent by the application prior to receipt of the change message are guaranteed to be delivered before the change message itself.[2]

- *External synchrony*. When a site updates its membership, all other sites are guaranteed either to have the same new membership or to be in a transition state to the new membership [RFJ93].

Partition Handling. Partition handling properties specify how the system behaves when a network partition occurs and when it is subsequently corrected. Policies for dealing with partitions can be divided into three phases: the policy used at the time the partition occurs (*partition time*), how operation proceeds while the sites are partitioned (*partitioned operation*), and how sites in separate partitions are merged when communication is reestablished (*partition join*). There are numerous options for each phase, with the following being common choices:

- *Individual notification*. A partition is treated as a sequence of individual site failures, with separate change messages delivered to the application for each site in other partitions (partition time).

- *Collective notification*. A single change message reporting all site failures is delivered to the application (partition time).

- *Majority operation*. Normal operation continues only in the partition with the majority of sites, if one exists (partitioned operation).

- *Continued operation*. Normal operation continues at all sites, with inconsistencies resolved at partition join time (partitioned operation).

- *Asymmetric join*. Partition join is reduced to sites in the minority partition simulating failure and joining the majority partition as recovering sites (partition join).

- *Collective join*. Partitions merge their memberships as one atomic membership change. For the memberships on all sites to be consistent after the merge, other membership change messages must be consistently ordered with respect to the merge message (partition join).

Note that, although collective join ensures that membership information is consistent at all sites following the merge, the problem of reconciling application states is the responsibility of the application level. This issue is eliminated in the asymmetric join policy, since members in the minority partition typically adopt the application state of the majority partition during recovery.

2.3 Composite protocols

Our configurable membership service is based on a model for constructing distributed services in which software modules referred to as *micro-protocols* are composed together to form a *composite protocol*. Composite protocols can then be combined with traditional network protocols using standard hierarchical techniques such as those supported by the *x*-kernel. The result is a two-level model that supports flexible interaction and data sharing between modules when necessary, but also allows the

[2]This property is similar to the property of the same name defined in [MAMSA94], but does not include all aspects of the functionality defined in that paper.

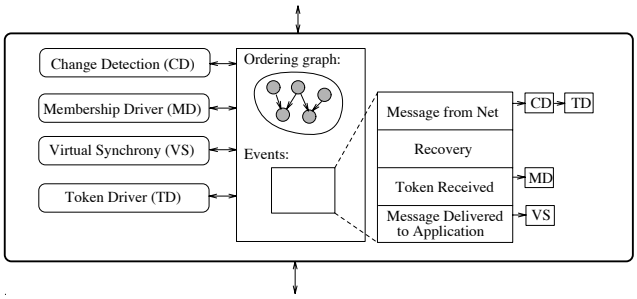

Figure 2: A composite protocol

strict hierarchical separation and proscribed interaction through a uniform protocol interface found in current hierarchical systems.

A micro-protocol is structured as a collection of *event handlers*. Each such handler is a procedure that is invoked when any of the *events* for which it is registered occurs. An event is the occurrence of a change in the system, such as the arrival of a message. An event may be triggered by the runtime system or by another micro-protocol depending on the situation. The invocation of the event handlers associated with an event can either be sequential, where all handlers are executed by a single thread in a specified order, or concurrent, where each handler is executed by its own thread. An event raised by a micro-protocol can either be blocking, where the invoker waits until all the handlers registered for the event have completed, or non-blocking, where the invoker continues execution without waiting.

Composite protocols are formed from a collection of micro-protocols configured together with a standard runtime system. This runtime system implements the event registration and triggering mechanism, and contains shared data (e.g., messages) that can be accessed by micro-protocols. A composite protocol presents the external interface of a simple protocol, which allows it to be combined with other simple or composite protocols. In our *x*-kernel prototype, for example, composite protocols are externally indistinguishable from standard *x*-kernel protocols.

This approach is depicted in Figure 2. In the middle is a composite protocol, which contains a shared data structure—in this case, an ordering graph containing the application and membership change messages—and some event definitions. The boxes to the left represent micro-protocols, while to the right are some common events with the list of micro-protocols that are to be invoked when the event occurs. Other protocols would typically be found both above and below this composite protocol, with messages being passed up or down through the protocol stack on their way between the application and the network.

The operations defined by the model for dealing with events are as follows.

- **register**(*event_name*,*handler_name*, *priority*). Notifies the runtime system that *handler_name* is to be executed when *event_name* is triggered. If the event is sequential, the handlers are executed in order according to *priority*.

- **trigger**(*event_name*,*arguments*). Notifies the runtime system that *event_name* has occurred. The runtime system will execute the appropriate handlers, passing *arguments* as invocation parameters.

- **deregister**(*event_name*,*handler_name*). Notifies the runtime system to remove the association between *event_name* and *handler_name*.

- **cancel_event()**. Instructs the runtime system to cancel further event handler invocations associated with the same event occurrence that caused the operation to be invoked. This operation is useful mostly for sequential events.

The model also supports a TIMEOUT event that is triggered by the passage of time. In this case, the *priority* parameter in the **register** operation is used to denote the time interval after which the specified handler is to be executed. Event handlers are usually persistent in the sense that they are invoked every time the specified event occurs until they are explicitly deregistered. The one exception is that event handlers registered for TIMEOUT are executed only once and then implicitly deregistered.

3 Design Overview

The basic components of a membership service built according to the model supported by composite protocols are events, messages, shared data structures, and micro-protocols. This section gives an overview of the first three; micro-protocols are described in the following section. First, however, we overview the general algorithmic strategy, which is based on token passing. Having a common algorithmic frame of reference is, of course, necessary to implement the properties described in section 2.2 as separate software modules that can be used together in various combinations.

3.1 Algorithmic strategy

Perhaps the key requirement for implementing many of the properties of membership is some means of information collection and dissemination. The various approaches for accomplishing this can be classified into three major categories: (1) broadcast based (e.g., [MPS93a, ADKM92b]), (2) coordinator based (e.g., [RB91, RFJ93]), and (3) token based (e.g., [RM89]). In examining each approach in light of our requirements, we selected the third based on the resulting simplicity of the micro-protocols.

The basic idea behind this approach is to organize the group members into a (logical) ring and then have a token that circulates around the ring. In contrast with other multicast and membership protocols that use token passing, the token in our scheme is used only for membership; regular communication need not be restricted to the ring or based on token passing. The role of the token is to collect and distribute the information required to realize the various properties. Specifically, the token has one record with multiple fields for each membership change underway at any given time. We call such a record a *membership entry*. Various micro-protocols exploit the information in the token in various ways.

Different properties of membership impose different requirements on how the token is used. For example, properties that involve primarily information dissemination, such as agreement, require that the token be rotated only once around the ring, while properties that also involve information collection, such as virtual synchrony, require that the token be rotated twice. The number of rotations actually used in a given configuration is the maximum of the number required across all micro-protocols that are included. The reliability of token passing is increased by requiring that the receiver acknowledge receipt of the token. Among other things, this strategy enables some aspects of failure detection to be integrated into the token passing mechanism.

To realize ordering-related properties, we use an ordering graph as described above, i.e., membership change messages are inserted into a graph of messages that constrains the delivery order to the application. We assume that actual delivery of messages from the graph to the application is handled independently from the membership service by the reliable communication component of the system,

which is configured within the same composite protocol. As already noted, we also assume that the underlying network provides asynchronous unreliable point-to-point message delivery, and that sites may suffer from crash or performance failures.

3.2 Events

As described in section 2.3, execution of code within micro-protocols is initiated when events occur. In general, events are used in this model for a variety of purposes, including to indicate a change of state in the composite protocol such as message arrival, to signal the opportunity to update shared variables or message fields, or as a procedure call to transfer control and data between two micro-protocols. Like signal variables in the monitor construct found in some concurrent programming languages, an event is a no-op if no handler is bound to that event. This feature de-couples micro-protocols to a certain degree and helps increase the configurability of the resulting system by removing the need to explicitly reference other micro-protocols.

In the membership service design, events can be classified into four categories:

- *Membership entry events*. For managing membership entries that circulate in the token.
- *Failure and recovery events*. For dealing with site failures, recoveries, and partitions.
- *Token handling events*. For dealing with token passing, regeneration, and merging after a partition.
- *Message handling events*. For managing application and membership change messages within the composite protocol.
- *Startup and restart events*. Two events, STARTUP_EV and RECOVERY_EV, that are generated at a site upon initial startup and recovery, respectively.

The first four are now described in more detail. For simplicity, all events are sequential and blocking.

Membership Entry Events. The most significant events in this category are FIRST_ROUND and SECOND_ROUND, which signal that a membership entry has been seen at the site for the first or second time, respectively. ADD_ENTRY is generated once current entries have been processed to signal the opportunity to add another entry to the token prior to it being passed to the next site. Similarly, NEW_ENTRY is generated when a new entry is added to allow various micro-protocols the opportunity to initialize fields of interest.

Failure and Recovery Events. Event SUSPECT_NEXT_DOWN is generated when the conditions for failure suspicion are met for the next site in the ring. In the case of a live membership service, this occurs when a certain number of token retransmissions are attempted without success; with an accurate service, this occurs when a message with a new incarnation number is received from a recovering site, indicating the earlier failure of the old incarnation. SUSPECT_CHANGE is a more general event indicating the suspected failure or recovery of any site in the system. POTENTIAL_ENTRY is generated to allow an opportunity to increase confidence in a suspected change or deny it prior to reporting it to the application. Finally, PARTITION is generated when a partition has been confirmed; note that since a partition cannot be distinguished from individual site failures at partition time, this event occurs after communication has been reestablished.

Token Handling Events. Event TOKEN_RECEIVED is generated when the token arrives at the site. FORWARDING_FAILED indicates that a particular attempt to pass the token to the next site in the ring has failed; this is also sometimes used as part of the failure detection process, as mentioned above. MERGE_TOKENS is generated when two tokens are merged into a single token, such as when partitions are merged.

Message Handing Events. Events are also used to manage application and membership messages. MSG_FROM_NET is generated when a message arrives at the composite protocol from lower-level protocols, while MSG_FROM_USER is generated when upper-level protocols pass along a message to be delivered to application processes on other sites. Both are triggered automatically by the runtime system. The event MSG_RECEIVED is generated within the membership service after some initial processing has been performed on every application message that arrives from lower-level protocols.

Events are also used to implement interactions between the membership and communication components of the composite protocol, as illustrated in Figure 3. Membership signals that a membership

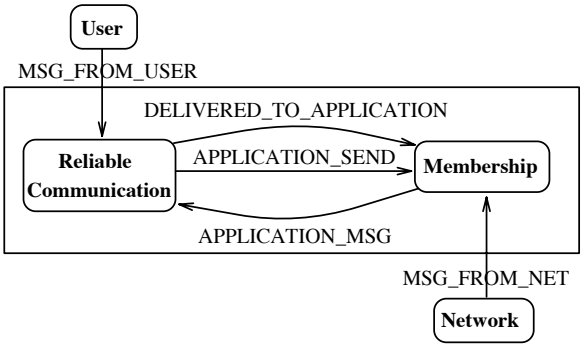

Figure 3: Interaction between components

change message is ready to be added to the ordering graph by triggering APPLICATION_MSG. The communication service indicates that an application message is ready to be sent to lower-level protocols with event APPLICATION_SEND, while DELIVERED_TO_APPLICATION is generated when a message has been delivered upwards towards the application. These latter two are often used by membership micro-protocols involved with ordering messages.

Note that the semantics of events make their use for component interaction qualitatively different than function calls. In this case, for example, some reasonable system configurations include no micro-protocols that field the event APPLICATION_SEND. If function calls were used, the communication code would have to be changed to avoid making this call, while with events, no change is needed since it can safely be generated with no effect. More importantly, since more than one handler can be bound to a single event, using this mechanism also makes it simple for multiple independent micro-protocols to be notified when an event occurs. In this case, for example, it is quite likely that multiple micro-protocols will field the APPLICATION_MSG event.

Finally, the event MEMBER_MSG is generated when a membership change message has been created, thereby allowing micro-protocols an opportunity to set fields in the message.

3.3 Membership change messages

Messages transmitted from the composite protocol up to the application process on a given site are either application data messages from other sites or membership change messages. As noted in section 2.1, membership change messages are the mechanism by which the membership service informs the application of site failures and recoveries so that it can, for example, update a local membership list. In certain cases, these messages are also used as control messages.

The membership change messages used in this design are the following:

- STARTUP indicates that the application can begin normal processing; includes a list of initial group members.
- SHUTDOWN indicates that the application should stop.
- FAILURE reports the failure of one site; if the site is this site, the application should stop.
- RECOVERY reports the recovery of one site; if the site is this site, the application can resume normal operation using the current membership information in the message.
- MERGE reports the merging of two partitions; the message contains the identities of the new group members.
- C_FAILURE reports the collective failure of more than one site, possibly due to a partition.
- PRE_MERGE indicates that the merging of two partitions is in progress; used to implement certain message ordering guarantees that require a site to stop sending messages until the merge is complete.
- PRE_CHANGE indicates that a change in membership is about to occur so that the application can alter behavior if necessary; for example, with some message ordering guarantees, the application must stop sending messages or change its membership into a transition state.

Note that the specific messages that an application may receive depends on the particular micro-protocols configured into the composite protocol. For example, a MERGE message will only be received if the micro-protocol that implements partition merging is included. Also, note that these are just the membership messages that are delivered to the application; other messages, such as those that implement token passing, are used by the membership service itself to communicate with peers on other sites.

3.4 Shared data structures

One of the most important benefits of using composite protocols is that it allows micro-protocols to share data. Our experience using the *x*-kernel to construct Consul [MPS93b] suggests that shared data facilitates implementation of modular fault-tolerant services, as does the experience reported independently by the developers of the xAMP atomic multicast suite [Fon94]. In the configurable membership service, the most important shared data structures are the following:

- **MsgGraph**. The ordering graph of messages.
- **Token**. The contents of the most recent token received at this site.

- **Membership**. An image of the application's view of the current membership.
- **ParList**. The membership service's current view of the group membership, i.e., the list of participants in the membership protocol; membership changes take effect earlier in **ParList** than **Membership**.
- **Delivered**. A vector with the identifier of the most recent message delivered from each site, which is used to determine when a message is eligible for delivery because all of its predecessors have been delivered; shared since other micro-protocols such as those concerned with recovery must be able to update it.
- **SuspectList**. All suspected membership changes that have not yet been reflected in **ParList**.

4 Micro-Protocols

This section describes the various micro-protocols that comprise the membership service, presenting the code for several. The goal is not to be exhaustive, but rather to give an overall view of how the service operates and some examples of the algorithmic and programming style used to write composite protocols. For expository purposes, we divide the micro-protocols into five categories:

- *Base micro-protocols*. Provide the base functionality needed by other micro-protocols, including message and token handling, and recovery.
- *Accuracy, liveness, and confidence micro-protocols*. Deal with detecting site failures and recoveries.
- *Agreement micro-protocols*. Implement the agreement process required for most variants of membership.
- *Ordering and synchrony micro-protocols*. Implement different varieties of message ordering guarantees.
- *Partition handling micro-protocols*. Implement different partition handling policies.

4.1 Base micro-protocols

The base micro-protocols are **MessageDriver**, **TokenDriver**, **Recovery**, and **StartUp**. The subsequent paragraphs summarize their functionality.

MessageDriver. This micro-protocol coordinates the traversal of application messages from the network to the application. It triggers MSG_RECEIVED when an application message arrives from the network and APPLICATION_MSG when the message is ready to be forwarded. Application messages sometimes have to be temporarily retained in the membership layer, for example, to implement virtual synchrony. To do this, **MessageDriver** provides a mechanism for releasing a message when all micro-protocols have finished operating on it. Specifically, two arrays are used: a global **Hold** array, which specifies which micro-protocols must execute for each message, and a corresponding **hold** array associated with each message, which specifies which micro-protocols have already executed. Thus, when **hold** is equivalent to **Hold**, the message can be forwarded.

The pseudo-code for **MessageDriver** is shown in Figure 4. Its general form is similar to most micro-

```
micro-protocol MessageDriver() {
    export procedure forward_up( var msg: ApplMessage, hold_index: int) {
        msg.hold[hold_index] = true;
        if for each i: Hold[i] = msg.hold[i] then trigger(APPLICATION_MSG,msg);
    }
    event handler msg_from_net(var msg:NetMessage) {
        if msg.type = DATA then { msg.amsg.hold[ ] = false;
            trigger(MSG_RECEIVED,msg.amsg); forward_up(msg.amsg,DEFAULT); }
    }
    event handler delivered_msg(var msg: ApplMessage) {
        if msg.type = FAILURE then Membership –= msg.changed;
        elseif msg.type = RECOVERY then Membership += msg.changed;
        . . . similar for other message types . . .
    }
    initial { Hold[DEFAULT] = true; register(MSG_FROM_NET,msg_from_net);
        register(DELIVERED_TO_APPLICATION,delivered_msg); }
}
```

Figure 4: **MessageDriver** micro-protocol

protocols: a few event handlers, initialization code, and possibly some local variables and functions. As can be seen from the pseudo-code, messages have a number of fields. These include its type (**type**), a unique identifier (**mid**), the sender (**sender**), the hold array mentioned above (**hold**), the message to be passed to the application (**amsg**), and, in the event of a membership change message, the identity of the failed or recovered site (**changed**). Not shown here, but used below, is an array of message identifiers (**pred**), which holds the predecessors of the message in the ordering graph.

TokenDriver. This micro-protocol's task is to implement for each partition the abstraction of an indestructible token that circulates among all functioning sites in the order dictated by the logical ring. In addition to the actual message passing involved in sending the token to the next site, much of the code in **TokenDriver** involves dealing with exceptional conditions such as lost token regeneration, site failures during regeneration, and the possibility of multiple tokens during the merging of partitions.

TokenDriver at a given site suspects that the token has been lost when it fails to receive it again a specified amount of time after passing it on. When this occurs, the micro-protocol first finds the most recent copy of the token that has been seen by any site in the ring. To facilitate this, **TokenDriver** at each site maintains a copy of the most recent token it has seen, together with its *version number* and *circulation count*; the former is incremented every time the token is regenerated, while the latter is incremented every time the site processes the token. The most recent copy of the token is discovered by circulating a *regeneration token*, which at any time holds the most recent copy seen so far. Once the regeneration token has circulated the entire ring once, it contains the most recent token, which is then used to create a new token. Multiple sites can—and often will—issue a regeneration token, but the algorithm ensures that only one site will have its regeneration token make the complete circuit.

Site failures may, of course, occur while the regeneration token is circulating. If this occurs, a membership entry will be added to the regeneration token and merged with the entries from the most recent copy of the lost token. New entries will not, however, be processed or any membership change messages forwarded to the application during the regeneration process

The **TokenDriver** micro-protocol triggers the event TOKEN_RECEIVED when a normal (i.e., not regeneration) token is received, MERGE_TOKENS when two tokens are merged as a part of token

regeneration, and FORWARDING_FAILED when the site to which a token is passed fails to acknowledge the reception within a specified time bound.

Recovery. This micro-protocol handles recovery of a site after failure. Its execution is initiated when the RECOVERY_EV event is triggered automatically by the runtime system upon recovery. At this point, **Recovery** begins sending JOIN messages to other sites, either using a previous membership list saved on stable storage or exploiting broadcast-based network hardware, if available. When another site receives this message, it triggers whichever recovery detection micro-protocol is configured into the composite protocol (see section 4.2), which begins the process of reintegrating the site back into the membership. The **MembershipDriver** micro-protocol (see section 4.3) manages the necessary agreement protocol, which involves adding a recovery entry to the token. While the token circulates, each site reads the recovery entry to obtain relevant information about the recovery, and updates it as needed with information needed to reinitialize the membership state of the recovering site. Based on what ordering properties are being enforced, the token will circulate either once or twice. Once a site has received the token the required number of times, it inserts a membership change message announcing the recovery into its ordering graph, with the specific location depending on the ordering properties.

The site that originated the recovery entry is also responsible for updating the membership state of the new member and inserting it into the logical ring. To do this, it sends a STATE message containing the relevant state of the membership service (e.g., the current membership list) and the token to the recovering site. Upon receiving this message, the **Recovery** micro-protocol updates its local data structures. It also uses information in the recovery entry of the token to initialize the state of other components of the composite protocol, such as the location in the ordering graph that the communication component should use as the starting point for forwarding messages up to the application. Once this has been done, the recovering site forwards the token to the next site and begins normal operation.

As is the case with token handling, the recovery and reintegration process is designed to tolerate site failures, additional recoveries, and similar exceptional conditions.

StartUp. This micro-protocol manages the startup process. When the STARTUP_EV event is generated by the runtime system, it sets the incarnation number, initializes the local copy of the token, and forwards the STARTUP message with the initial membership to the application.

4.2 Accuracy, liveness, and confidence micro-protocols

For detecting site failures, micro-protocols that implement both live and accurate algorithms are provided as alternatives. Live detection is based on lack of response from a site, i.e., timeouts. Accurate detection, on the other hand, cannot be based on communication since the network is assumed to be asynchronous. As a result, our implementation, like that described in [OIOP93] for Mach, detects a site failure only when the failed site recovers and reestablishes communication. Similarly, accurate recovery detection—the only kind possible in asynchronous systems—is implemented by the recovering site contacting other sites upon recovery as described in the previous section. The following micro-protocols implement these properties:

- **LiveFailureDetection**. Triggers event SUSPECT_NEXT_DOWN signaling a suspected site failure if token retransmission fails a specified number of times. Triggers SUSPECT_CHANGE instead if a site that is expected to communicate for some other reason does not respond in a timely

```
micro-protocol LiveFailureDetection(LIMIT:int, check_period: real) {
    var SilentList: set of int;    % list of sites not heard of lately

    event handler handle_failure(var site:int, attempts:int) {
        if attempts < LIMIT and (site,FAILURE) ∉ SuspectList then attempts++;
        else { SuspectList += (site,FAILURE); attempts = 1;
            trigger(SUSPECT_NEXT_DOWN,site); }
    }
    event handler handle_msg_from_net(var msg: NetMessage) {
        if msg.type != JOIN then {
            if (msg.sender,FAILURE) ∈ SuspectList then
                SuspectList –= (msg.sender,FAILURE);
            if msg.sender ∈ SilentList then SilentList –= msg.sender; }
        elseif msg.sender ∈ Membership and (msg.sender,FAILURE) ∉ SuspectList then {
            SuspectList += (msg.sender,FAILURE);
            trigger(SUSPECT_CHANGE,msg.sender,FAILURE); }
    }
    event handler handle_new_membership_msg(msg:ApplMessage) {
        if msg.type == FAILURE then SuspectList –= (msg.changed,FAILURE);
    }
    event handler monitor() {
        for each m:int ∈ Membership do {
            if m ∈ SilentList and (m,FAILURE) ∉ SuspectList then {
                SuspectList += (m,FAILURE);
                trigger(SUSPECT_CHANGE,m,FAILURE); }
            SilentList += m; }
        register(TIMEOUT,monitor,check_period);
    }
    initial { register(FORWARDING_FAILED,handle_failure);
        register(MSG_FROM_NET,handle_msg_from_net);
        register(MEMBER_MSG,handle_new_membership_msg);
        register(TIMEOUT,monitor,check_period); }
}
```

Figure 5: **LiveFailureDetection** micro-protocol

manner, or if a site that is already in the membership list attempts to join. The pseudo-code for **LiveFailureDetection** is shown in Figure 5.

- **AccurateRecoveryDetection**. Triggers SUSPECT_CHANGE signaling a suspected site recovery upon receiving a message from a site not currently in the membership. Used in combination with **LiveFailureDetection**.

- **AccurateDetection**. Implements accurate detection of both site failures and recoveries. Triggers SUSPECT_CHANGE signaling a suspected site failure and succeeding recovery when a message arrives with an incarnation number greater than the current incarnation number for that site. Also inserts the current incarnation number in outgoing messages.

Our design supports two versions of the confidence property. The first is single site suspicion, where no confirmation is needed from other sites. In this case, a suspected membership change can simply be entered into the token and circulated among all group members. The second is a voting-based process, which is implemented by the micro-protocol **VotedDecision**. When the event POTENTIAL_ENTRY is triggered, **VotedDecision** sends out a request for votes and sets a timer using the facilities for TIMEOUT events provided by the runtime system. When the timer expires, all votes that have been received

are examined. If any site has responded "no", the result in negative, that is, the conclusion is that no membership change has occurred. Otherwise, the result is positive. Individual sites base their responses on whether or not the suspected site is in their **SuspectList**. Many other variants of voting-based policies are, of course, possible.

4.3 Agreement micro-protocols

Implementing agreement on site failures and recoveries is straightforward given the abstraction of an indestructible token. In particular, since the token is guaranteed to be received periodically by every operational site, all that is required is to enter the change in the token and circulate it. Sites then read the entry when the token arrives and deliver a membership change message to the application at the appropriate place in the message stream.

The micro-protocol **MembershipDriver** implements agreement and coordinates the overall execution of the membership protocol. It triggers events ADD_ENTRY, FIRST_ROUND, SECOND_ROUND, and MEMBER_MSG. It also maintains information about the number of rotations needed for each membership entry in the token. Special attention is paid to entries whose *reporter*—the site that originally added the entry to the token—fails during execution of the protocol. This situation is handled by having such entries be "adopted" by other sites, which then behave as the reporter for the remainder of the protocol.

A second membership driver micro-protocol called **SimpleMembershipDriver** is provided as an option for applications not requiring agreement. Rather than circulate information in the token, it simply translates local detection of failures and recoveries into membership change messages that are delivered to the application. It also implements a simple recovery facility for this type of application.

All remaining micro-protocols assume that **MembershipDriver** is configured into the system.

4.4 Ordering and synchrony micro-protocols

The **MembershipDriver** micro-protocol implements FIFO ordering of membership change messages as a free side-effect of the agreement process. Other orderings are implemented by separate micro-protocols, as follows

Total order. Total ordering of membership change messages is implemented by simply forwarding membership change messages to the application in the order the changes are recorded in the token. Since every site sees the same token, every site delivers the messages in the same total order using only one round of token rotation.

The **TotalOrder** micro-protocol (Figure 6) implements this property by translating the ordering of entries in the token into a total order in the ordering graph using message predecessor fields. Note that the strong guarantees provided by **TokenDriver** and **MembershipDriver** greatly simplify this micro-protocol.

Agreement on Last Message. Properties such as agreement on last message that require ordering membership change messages with respect to application messages are somewhat more complex. For example, the **AgreedLast** micro-protocol implements this property by collecting information in the token about the last message received from the failed site. This information, which is stored in the membership entry, is updated at a site if that site has received a message with a higher identifier than the one currently in the token. After one rotation, then, the token holds the identifier of the most recent

```
micro-protocol TotalOrder() {
    var previous_mid: int;    % id of the previously processed msg in total order

    event handler handle_mship_msg( var msg:ApplMessage,entry:EntryType) {
        msg.pred[ ] += previous_mid; previous_mid = msg.mid; }

    initial { previous_mid = 0; register(MEMBER_MSG,handle_mship_msg); }
}
```

Figure 6: **TotalOrder** micro-protocol

message that any site has received from the failed site at the time it updated the entry. This message is taken to be the agreed-upon last message, and the token rotated a second time to disseminate the result. After receiving the token a second time, each site places the appropriate membership change message in the ordering graph immediately after the agreed-upon last message. Note that during this process, delivery of application messages from the suspected failed site must be stopped.

Figure 7 presents the pseudo-code for **AgreedLast**.

Other Ordering Micro-protocols. A variety of other message-ordering options are provided by the micro-protocols **AgreedPred**, **AgreedSucc**, **VirtualSynchrony**, **ExtendedVirtualSynchrony**, and **ExternalSynchrony**. Each realizes the ordering guarantee in section 2.2 corresponding to its name. The algorithms used to implement these properties are similar to that used in **AgreedLast**, with the differences being the type of information collected on the first round, how this information is used, and whether message delivery to the application can continue during the process. Details can be found in [Hil96].

4.5 Partition handling micro-protocols

As noted in section 2.2, the policies that dictate how a system operates in the presence of partitions can be divided into three phases: partition time, partitioned operation, and partition join. The micro-protocols relevant to each phase are described below.

Partition Time. By default, the membership service implements individual notification, where the membership changes associated with a partition are treated as individual site failures. The alternative collective notification policy is provided by the **CollectiveNotification** micro-protocol, which reports the failure of all sites in other partitions in a single membership change message.[3] It does this by waiting for the NEW_ENTRY or MERGE_TOKENS events, and then when they occur, combining all failure entries in the token into a single entry. The entries are combined so that the ordering properties guaranteed for the combined entry are inherited from the first entry in the token. Once the entries are combined, the token is circulated again to ensure that every site sees the combined entry. To guarantee that no site generates a membership change message for an entry before **CollectiveNotification** has a chance to combine entries, each entry is circulated at least once around the ring before a membership change message is delivered to the application. This also guarantees that all sites in other partitions are included in the collective entry.

[3] Note that simultaneous true site failures will also be reported collectively since such situations are impossible to distinguish from partitions in asynchronous systems.

```
micro-protocol AgreedLast() {
    var LastSeen[ ], LastAllowed[ ]: int; mutex: semaphore;

    event handler first_round(var entry:EntryType) {
        var s: int;
        if entry.type == FAILURE then { P(mutex); s = entry.changed;
            entry.pred[s] = max(LastSeen[s],entry.pred[s]);
            LastAllowed[s] = entry.pred[s]; V(mutex); }
        else if entry.type == C_FAILURE then { P(mutex);
            for each s ∈ entry.members do { entry.pred[s] = max(LastSeen[s],entry.pred[s]);
                LastAllowed[s] = entry.pred[s]; }
            V(mutex); }
    }
    event handler new_membership_msg( var entry:EntryType, msg: ApplMessage) {
        var s: int;
        if entry.type == FAILURE then { P(mutex); s = entry.changed;
            msg.pred[s] = max(msg.pred[s],entry.pred[s]);
            LastAllowed[s] = msg.pred[s]; V(mutex); }
        else if entry.type == C_FAILURE then { . . . similar to above . . . }
    }
    event handler handle_delivered_msg(var msg:ApplMessage) {
        if msg.type == FAILURE then { P(mutex);
            LastAllowed[msg.changed] = MAXINT; V(mutex); }
        else if msg.type == C_FAILURE then { . . . similar to above . . . }
    }
    event handler handle_msg(var msg:ApplMessage) { P(mutex);
        if LastAllowed[msg.sender] < msg.mid then { V(mutex); cancel_event(); }
        else { LastSeen[msg.sender] = max(LastSeen[msg.sender],msg.mid); V(mutex); }
    }
    initial { register(FIRST_ROUND,first_round); register(MSG_RECEIVED,handle_msg);
        LastSeen[ ] = 0; register(DELIVERED_TO_APPLICATION,handle_delivered_msg);
        LastAllowed[ ] = MAXINT; register(MEMBER_MSG,new_membership_msg); }
}
```

Figure 7: **AgreedLast** micro-protocol

Partitioned Operation. The policy for what level of service a system should provide during partitions is inherently an application decision, so the membership service generally only provides supporting information. In our design, this is done by the **AugmentedNotification** micro-protocol, which augments each membership change message with majority/minority status information depending on whether the site is in the majority partition or not. Note that the appropriate value is simple to calculate, assuming that the maximum membership of the group is known and that membership change messages are delivered in total order. The application can use this information, for example, to halt processing or reduce the level of service in minority partitions.

Partition Join. The merging of partitions is initiated by the **PartitionDetection** micro-protocol, which attempts to detect the existence of other partitions by periodically sending "I am alive" messages containing the current membership to all sites that are currently considered failed. Upon receipt of such a message at some site *S*, the event PARTITION is triggered if the membership list in the message has no sites in common with the membership list on *S*. If the lists do overlap—which might occur, for example, if the partition was of such short duration that the process of removing sites from the list was incomplete in one or both partitions—then the sites in the intersection are removed from the membership of *S*'s

```
micro-protocol AsymmetricJoin() {
    var OtherPartition: set of int;    % sites in the other partition

    event handler handle_partition(Members: set of int) {
        if dominate(Members,ParList) then {
            OtherPartition = Members; register(ADD_ENTRY,enter_shutdown); }
    }
    event handler enter_shutdown(var token: TokenType) {
        var entry: EntryType;
        if SHUTDOWN ∉ token.entries then {
            entry.type = SHUTDOWN; entry.members = OtherPartition;
            token.entries += entry; trigger(NEW_ENTRY,entry); }
        deregister(ADD_ENTRY,enter_shutdown);
        register(DELIVERED_TO_APPLICATION,handle_delivered_msg);
    }
    event handler handle_delivered_msg(msg: ApplMessage) {
        if msg.type == SHUTDOWN then {
            deregister(DELIVERED_TO_APPLICATION,handle_delivered_msg);
            status = DOWN; shutdown_and_restart(msg.members); }
    }
    initial { register(PARTITION,handle_partition); }
}
```

Figure 8: **AsymmetricJoin** micro-protocol

partition prior to beginning the merge process.

Two alternative micro-protocols are provided for implementing the partition merge when PARTITION is triggered, **CollectiveJoin** and **AsymmetricJoin**. The first combines two partitions into a single one, including merging the two logical rings used for communication. This is accomplished by one partition giving up its token to a site in the other partition, which then combines the tokens into a single token as described in section 4.1. A special membership change entry of type MERGE is then inserted into the token and circulated to inform all sites in the combined membership of the new membership information. **AsymmetricJoin** handles partition join by forcing sites in the minority partition to fail prior to being allowed to rejoin the majority partition as individual recovering sites. The shutdown is coordinated by adding a SHUTDOWN entry to the token in the minority partition.

The pseudo-code for **AsymmetricJoin** is shown in Figure 8. Functions *dominate* and *shutdown_and_restart* are defined elsewhere. The former defines the dominance of two partitions based on group size and site identifiers, as discussed above. The latter, which is executed only by sites in the non-dominant partition, takes the assumed membership of the dominant partition as argument and simulates the failure and restart of the composite protocol. It also triggers the event RECOVERY_EV.

Finally, the **ExtendedWithPartition** micro-protocol implements extended virtual synchrony between application messages and membership change messages reporting partition merges, similar to that defined in [MAMSA94]. This micro-protocol is distinct from **ExtendedVirtualSynchrony** since the predecessor sets of such messages are different in the sites in the two merging partitions.

Other partition handling micro-protocols. In numerous membership services [Cri91, HS95b, KT91, KGR91, MSMA94, MPS93a, RB91], it is simply assumed that partitions will not occur, or that only one partition will continue to operate. The **OnePartition** micro-protocol implements a simple strategy that approximates this behavior. In particular, when any message other than JOIN is received from a site outside the current membership, **OnePartition** sends a STOP message that forces the offending site

to halt. A simple dominance relationship based on group size and maximum site identifier is used to ensure that only one partition remains active.

5 Configuring a Custom Membership Service

The collection of micro-protocols outlined above provides the basis for building a membership service with properties customized to the needs of a given application. In principle, those micro-protocols that provide the desired properties are combined at system configuration time to construct an instance of the service. However, as might be expected, not all combinations are feasible, largely due to functional dependencies between micro-protocols. Here, we discuss the issue of dependencies in the context of the design presented in previous sections and give a *configuration graph* that summarizes the legal combinations.

5.1 Dependencies

A micro-protocol m_1 is said to *depend on* micro-protocol m_2 if m_2 must be present in the system and provide its specified service in order for m_1 to provide its specified service. In practice, this means that if micro-protocol m_1 is configured into a system, micro-protocol m_2 must be configured into the system as well. Several examples of dependencies were given in the micro-protocol descriptions above. For example, in section 4.4, we noted that **TotalOrder** builds on the guarantees provided by **MembershipDriver** and **TokenDriver**. Such assumptions are indicative of dependencies between micro-protocols.

In general, there are a number of possible sources of dependencies between micro-protocols. One is the inherent relationship between the properties that are being implemented. For example, totally ordering membership change messages is impossible unless such messages are present on all sites, so the property of total ordering depends on agreement. Thus, in our design, this means that **TotalOrder** depends on **MembershipDriver**. A complete evaluation of dependencies between properties related to membership can be found in [HS95c, Hil96].

Another source of dependencies comes from the way in which properties are implemented, rather than any inherent relationship between them. That is, although it might be possible to implement properties p_1 and p_2 independently, it is easier to implement p_1 assuming that p_2 is guaranteed, leading to a dependency between the micro-protocols implementing the properties. For example, it is easier to implement many of the ordering properties with respect to application messages given the assumption that membership change messages are totally ordered. In our design, however, we purposely avoided such dependencies to achieve maximum configurability.

The implication of the above discussion is that it usually requires a close semantic evaluation of properties and their implementations to determine the full set of dependencies in a given design. However, some dependencies can be identified syntactically based on how interactions between micro-protocols are programmed using either events or shared variables. For example, the **AugmentedNotification** micro-protocol used for partition handling relies on the membership change messages being ordered prior to being delivered to the application, which **TotalOrder** implements by setting the message predecessor fields in an appropriate manner. Thus, in this case, the ordering graph is the shared variable that identifies the dependency. As another example, the dependencies between various ordering micro-protocols and **MembershipDriver** can be identified by events that are triggered by **MembershipDriver** and fielded by the ordering micro-protocols.

5.2 Configuration graph

Configuration graphs are a graphical method for representing configuration constraints caused by dependencies between micro-protocols. Such a graph is a directed graph, $G = (N, E)$, where the nodes N represent micro-protocols and the edges E represent dependencies. Often, a micro-protocol m_1 requires that any one of a set of micro protocols, $\{m_2, m_3, \ldots\}$, be present. This is represented in the dependency graph by grouping $m_2, m_3, \ldots$ together and having an edge from m_1 to this group.

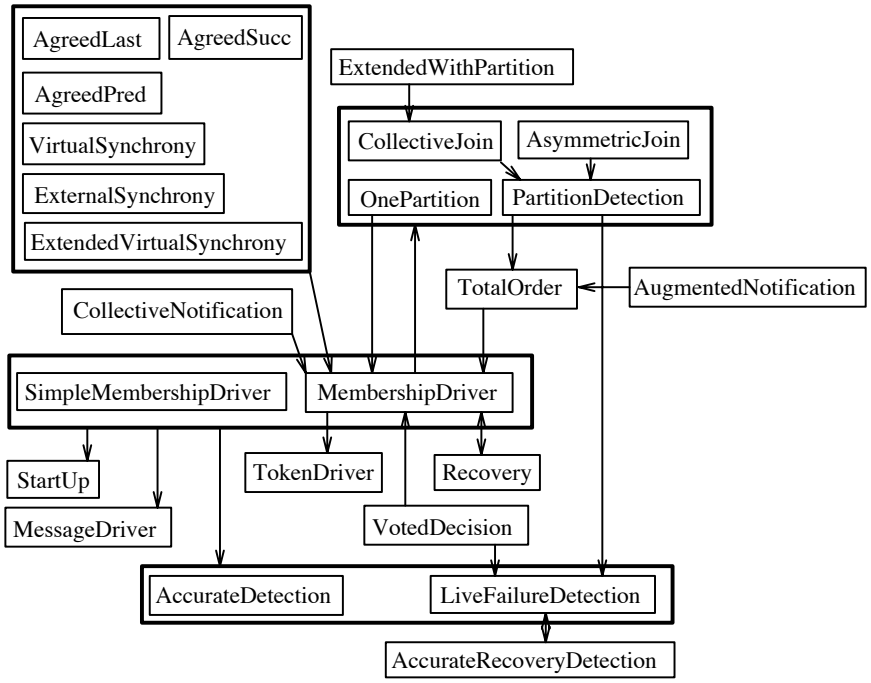

Figure 9: Configuration graph

A configuration graph can be viewed as a tool for configuring a customized service. The designer of a system first decides which service properties are required and identifies the micro-protocols that implement those properties in the configuration graph. These micro-protocols are then included in the system, along with all micro-protocols on which the chosen ones depend. Essentially, this implies that all micro-protocols in the transitive closure of those implementing the selected properties must be included.

Figure 9 gives the configuration graph for the membership micro-protocols discussed in this paper. Using the graph we can, for example, configure a simple membership service consisting of the micro-protocols **MessageDriver**, **SimpleMembershipDriver**, **AccurateDetection**, and **StartUp**. This service provides accurate but uncoordinated membership change indications to the higher level protocols. An example of a more complicated membership service would be one that provides virtual synchrony, and handles network partitions by allowing computation to continue in each partition and then combines partitions by forcing sites in one to fail and rejoin as individual members. Such a service consists of the micro-protocols **AsymmetricJoin**, **PartitionDetection**, **VirtualSynchrony**, **TotalOrder**, **MembershipDriver**, **TokenDriver**, **MessageDriver**, **Recovery**, **StartUp**, **AccurateRecoveryDetection**, and **LiveFailureDetection**. In this manner, over 750 semantically distinct membership services can be

configured from the micro-protocols in this collection.

6 Prototype Implementation

6.1 Overview

A prototype of the configurable membership service has been implemented using C++. The implementation consists of approximately 9000 lines of code and uses the Sun Solaris operating system's thread package to implement event handling and other control aspects of the runtime system. Currently, the multiple sites of a distributed architecture are simulated within a single address space, although the code for all the micro-protocols and much of the runtime system would carry over unchanged to a true distributed implementation using C++. Using a simulated environment as an initial step has, of course, numerous advantages. For example, it facilitates rapid prototyping of micro-protocols since it is easier to execute test runs and collect results. It also makes it possible to control execution parameters to a degree not possible in a real system, including the number of sites, the message transmission times, and failure rates.

As noted in section 1, our eventual goal is to implement this service using an *x*-kernel based system currently under development [BS95]. This system augments the standard hierarchical model of protocol composition implemented by the *x*-kernel with support for micro-protocols and event-driven execution. The system currently runs on DecStation 5000 workstations connected by Ethernet, and has been used to implement a micro-protocol suite for group RPC.

6.2 Software organization

The overall software organization resembles Figure 1, with C++ classes being divided into three major portions that implement that application, network, and communication layer, respectively. The application is simulated by class `User`, which generates application messages. It also logs application and membership change messages received from the communication service for debugging purposes. One object of this class is created for each site in the simulated system.

The network is simulated by class `Network`, which implements the abstraction of an unreliable point-to-point and multicast communication medium. In addition to routing messages between sites, `Network` implements a short transmission delay and generates communication failures by deciding for each message and destination whether or not to deliver the message based on a random number. Site failures are simulated by shutting down all micro-protocols in a controlled manner and zeroing out appropriate data structures. Network partitions are simulated by maintaining a table that specifies the partition for each site, so that a message from a given site is only delivered to a destination if it resides in the same partition. This table can be altered at runtime to simulate the creation and joining of partitions. A single `Network` object is created for each simulation.

Most of the rest of the C++ classes implement the communication layer, including the communication service, the configurable membership service, and the infrastructure required to implement composite protocols. Perhaps the two most important are the base classes `CompositeProtocol` and `MicroProtocol`. `CompositeProtocol` contains the runtime system of composite protocols, implementing the event-driven execution model and providing such operations as the registering and deregistering of events. It also implements the interactions with the application and network layers, such as providing operations for those layers to transfer messages to the composite protocol. This class also contains the code that triggers system-defined events, such as MSG_FROM_NET and MSG_FROM_USER.

Derived classes are defined from `CompositeProtocol` for each specific type of composite protocol, with object instances being created from these for each simulated site.

`MicroProtocol` is the base class from which the micro-protocols implementing the functionality of the communication and membership services are defined. In particular, each micro-protocol is a derived class based on `MicroProtocol`, which allows them to be dealt with as uniform objects whenever possible. A derived class is provided for each membership micro-protocol described in section 4. The communication service, which implements reliable causally ordered multicast and the ordering graph abstraction, is also defined in this way. Object instances are created for each site according to the specific configuration desired.

6.3 Testing and experience

The C++ prototype has been used to test a variety of different membership services. Given the large number of micro-protocol combinations possible based on the configuration graph in figure 9, the services tested were representative rather than exhaustive. First, all possible combinations that did not involve any of the micro-protocols that implement ordering properties with respect to application messages were tested. Then, to test these micro-protocols—**AgreedLast**, **AgreedSucc**, **AgreedPred**, **VirtualSynchrony**, **External Synchrony**, and **ExtendedVirtual Synchrony**—each was combined with five representative configurations of the remaining micro-protocols, one with **AccurateDetection** and four with **LiveFailureDetection** plus different ways of dealing with partitions. The test suite included scenarios involving multiple failures and recoveries, network partitions, and token loss, as well as normal processing.

Testing was performed as a black-box process in which various output results were monitored for a given set of inputs. For a membership service, the primary determinants of correctness are the messages received by the application level on each site and their ordering, so message logs maintained in `User` objects were the main source of information. Token passing was also traced to validate `TokenDriver`, arguably the most complicated micro-protocol. Execution of other individual micro-protocols could also be traced by setting the appropriate bit in a tracing mask; this causes event handling code in the micro-protocols to generate trace information every time one of its event handlers is invoked.

Building a version of the configurable membership service in this simulated environment has demonstrated several things, in our view. One is the overall viability of our modularization approach, where properties are mapped directly to fine-grained software modules to enhance configurability and customization. Another is the value of event-driven execution for decoupling modules and thereby minimizing the software changes needed to support configurability. The property-based modularity and configurability of the service also turned out to be an asset during the testing process itself. For example, during debugging, it was often easy to identify the property that was not being properly enforced, which automatically identified the offending micro-protocol. The ability to include or exclude micro-protocols easily also helped narrow the collection of modules that had to be examined when other types of problems occurred.

Of course, one question that cannot be answered from a simulation of this type concerns the performance cost, if any, associated with our approach to modularization. While a final resolution of this issue must await the completion of the *x*-kernel based version of the service, we note that initial experiments with a similar implementation of group RPC show modest execution overhead [BS95]. Other experiments using the *x*-kernel also suggest that fine-grained modularity need not imply a serious performance penalty [OP92].

7 Discussion

7.1 Micro-protocol execution costs

This choice of which micro-protocols to include when building a customized membership service is based primarily on the functional guarantees required by the application. However, another consideration is the incremental execution cost associated with guaranteeing the associated property, such the number of extra messages required and the additional delay that it imposes on the delivery of messages to the application. To examine these costs relative to the micro-protocols discussed above, we first divide the functionality implemented by the membership service into two phases:

1. *Detection*: Initial detection of suspected site failures and recoveries.
2. *Coordination*: Subsequent processing required for sites to coordinate decision and deliver membership change message to the application.

Each phase incurs separate execution overhead based on the specific properties being enforced.

For the detection phase, the metric of interest is *detection delay*, that is, the time it takes from when a change occurs until the initial suspicion is signaled at some site. For recovery detection, both **AccurateRecoveryDetection** and **AccurateDetection** are based on receiving a JOIN message from the recovering site, so the detection delay depends on how quickly this message is sent and received after the site restarts. For failure detection, the delay depends on whether **AccurateDetection** or **LiveFailureDetection** is used. **AccurateDetection** only detects a failure once the failed site recovers, so the detection delay may be arbitrarily long. On the other hand, as with most live failure detection schemes, the delay associated with **LiveFailureDetection** depends on the frequency of message exchange and the timeout interval used before a suspicion is signaled. In our particular design, detection delay can be reduced either by circulating the token faster, which increases the network load, or by altering the token passing protocol to reduce the maximum number of retransmissions or shrink the timeout interval, both of which increase the probability of false failure suspicions. **VotedDecision** reduces the probability of false detections at the cost of increased detection delay.

For the coordination phase, the metrics of interest are *agreement cost* and *delivery delay*. Agreement cost is the number of messages it takes to collect and distribute information about a membership change to all sites so that the selected properties are guaranteed. Given our token-based protocols, the agreement cost can most easily be analyzed in terms of how many token rotations a specific property requires. For the micro-protocols in our service, these costs are:

- *One Rotation*. **TotalOrder**, **AugmentedNotification**, **Recovery**, and **AsymmetricJoin**.
- *Two Rotations*. All other ordering micro-protocols and **CollectiveJoin**.
- *Between Two and Three Rotations*. **CollectiveNotification**.

Delivery delay is the extra delay imposed by the membership service on the time it takes a message to be delivered to the application. Although it is difficult to calculate such delays in absolute terms, examining the relative delays between micro-protocols can be instructive. The delays for the relevant micro-protocols are as follows:

- **AgreedPred**. None, since algorithm will construct an agreed predecessor set consisting of messages already delivered on every site.

- **AgreedLast**. Halts delivery of messages from suspected site during agreement, and requires delay of membership change message until agreed last message is delivered.
- **AgreedSucc**. Halts delivery of all messages during agreement, but does not delay membership change message.
- **VirtualSynchrony**. Halts delivery of all messages during agreement, and requires delay of membership change message until agreed predecessor set is delivered.
- **ExternalSynchrony**. No extra ordering delay, but requires each site to deliver PRE_CHANGE message during first token rotation prior to forwarding token to next site.
- **ExtendedVirtualSynchrony**. Same as **ExternalSynchrony** for token passing and **VirtualSynchrony** for membership change messages, except that extra delay may be incurred since agreed predecessor set consists of all messages sent before the initiation of agreement.

As would be expected, the delays are roughly proportional to the strength of the guarantees provided.

7.2 Related work

Membership services and protocols have been the subject of a large number of papers. Some of the work has been based on a synchronous system model, where bounds are placed on the network transmission time [Cri91, EL90, KGR91, LE90, SCA94]. Other work assumes an asynchronous model similar to that used in this paper [ADKM92a, AMMS+95, DMS94, EL95, GT92, MPS92, MPS93a, MAMSA94, RB91, SM94]. Unlike our configurable service, however, all these services guarantee only a single collection of properties, or at most, offer a small number of choices.

Schemes based on logical rings or token passing are used by many multicast, membership, and system diagnosis protocols. For example, token passing is used as a means of implementing reliable totally ordered multicast in the Reliable Broadcast Protocol [CM84], Token-Passing Multicast (TPM) protocol [RM89], Multicasting Transport Protocol (MTP) [AFM92], Totem [AMMS+95], Pinwheel [CM95], and Reliable Multicast Protocol (RMP) [WMK95]. In these protocols, the site possessing the token is either the only site that is allowed to send a message or the site that assigns a global ordering to messages sent by all sites. Most of these protocols deal with site failures and recoveries, as well as the possibility of token loss. With the exception of TPM, however, all deal with membership changes using broadcast-based schemes that are independent of the token passing used during normal processing; in these cases, the token is recreated only after agreement has been reached on the new membership.

The algorithms used in TMP are perhaps closest to those used in our membership service, especially its use of the token to recreate group membership after a failure. When a site suspects that a token loss or site failure has occurred, it generates a *recovery token* and circulates it to collect the identities of operational sites. Multiple recovery tokens are eliminated by having each site only forward a token if it was created by a site with a larger identifier. After agreement on the new membership has been reached, a *clean-up token* is circulated to collect and disseminate information about the messages received on each site so that missing messages can be requested. This token also collects the maximum sequence number across all delivered messages as it circulates, which is then used to initialize the new token during recovery.[4]

Although the TMP protocol employs a token to collect and disseminate information in much the same way as we do, the underlying algorithms are quite different. For example, unlike TPM, in our

[4] Totem uses a *commit token* for a similar purpose [AMMS+95].

approach, information about concurrent membership changes—including partitions—is collected and disseminated using a single token. This difference changes many of the details of token handling, such as the steps taken to regenerate the token when failures occur. Furthermore, our design emphasizes configurability and facilitates the construction of customized membership services, rather than implementing a single set of properties as does TMP.

A number of membership and system diagnosis protocols organize sites into a logical ring structure without using a token. For example, the protocols in [RFJ93] use a ring to detect membership changes by having each site monitor its neighbor. Once a failure or recovery is detected, however, a membership protocol that employs a coordinator process is employed rather than using the ring. Some system diagnosis protocols, such as the Adaptive DSD protocol [BB91], use a logical ring for failure monitoring and information propagation. A simple membership protocol derived from Adaptive DSD that also uses a ring organization but not a token is introduced in [Hil95].

In addition to research on specific membership protocols, three other projects have investigated issues similar to those addressed in this paper. The membership service in Horus, a system for constructing modular networking software according to a hierarchical model [RBG+95, RB95], provides some degree of choice in the properties that it guarantees. In particular, the application may choose to deal with partitions by using a single partition approach or by allowing computation to continue in all partitions. In [RFJ93], a family of three membership protocols are described, where each protocol provides different guarantees. Although not configurable in the same sense as ours, the motivation—that different applications need different guarantees—is similar.

Finally, in [SR93], membership services are divided into three components: a Failure Suspector, a Multicast Component, and a View Component. The Failure Suspector implements the equivalent of the change detection property in our approach, while the Multicast Component provides reliable communication with virtually synchronous message ordering. Unlike our approach in which multicast and membership are separate, however, their Multicast Component combines these two functions. The View Component implements the equivalent of our agreement property by guaranteeing that all sites have the same view of the membership. While similar in that they decompose membership services into components, our approach provides a richer classification of properties and extends the concept to an actual software system supporting a high degree of configurability and customization.

8 Conclusions

The modular membership service described in this paper facilitates the construction of a customized fault-tolerance support layer that can provide the specific execution guarantees needed by a given application. By supporting this type of customization and configuration, our approach reduces the size and complexity of the system, thereby increasing the likelihood that it will operate as intended. It also has the potential to improve application performance by giving the designer explicit control over the tradeoff between the strength of the guarantees provided and the performance; rather than having to accept guarantees stronger than needed and thereby incur extra execution costs, the designer can select—and pay for—only those guarantees that are truly required. The approach is based on mapping abstract properties to individual micro-protocols, which are then configured together with a standard runtime system to form a composite protocol. The mapping of abstract properties to software modules and the event-driven model supported by the runtime system both enhance the overall configurability of the resulting system.

Our future work involving configurable services will concentrate in several areas. As already noted, one is porting the membership service to the *x*-kernel based system under development, which

will facilitate investigation of issues such as performance. Another is further work on applying our approach to other services that simplify the construction of fault-tolerant distributed applications, such as atomic multicast, group RPC, and distributed transactions. Finally, we plan to extend our research to include distributed services that provide other types of execution guarantees, such as those involving real time and security. All these investigations are related to our overall goal of developing a unifying architectural framework for building configurable support software for a wide variety of highly dependable applications.

References

[ADKM92a] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Membership algorithms for multicast communication groups. In *Proceedings of the 6th International Workshop on Distributed Algorithms (Lecture Notes in Computer Science 647)*, pages 292–312, Haifa, Israel, Nov 1992.

[ADKM92b] Y. Amir, D. Dolev, S. Kramer, and D. Malki. Transis: A communication sub-system for high availability. In *Proceedings of the 22nd International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, Jul 1992.

[AFM92] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. Request for Comments (Informational) RFC 1301, Internet Engineering Task Force, Feb 1992.

[AMMS+95] Y. Amir, L. Moser, P. Melliar-Smith, D. Agarwal, and P. Ciarfella. The Totem single-ring ordering and membership protocol. *ACM Transactions on Computer Systems*, 13(4):311–342, 1995.

[BB91] R. Bianchini and R. Buskens. An adaptive distributed system-level diagnosis algorithm and its implementation. In *Proceedings of the 21st International Symposium on Fault-Tolerant Computing*, pages 222–229, Jun 1991.

[BG93] K. Birman and B. Glade. Consistent failure reporting in reliable communication systems. Technical Report 93-1349, Department of Computer Science, Cornell University, May 1993.

[BS95] N. T. Bhatti and R. D. Schlichting. A system for constructing configurable high-level protocols. In *Proceedings of SIGCOMM '95*, pages 138–150, Cambridge, MA, Aug 1995.

[BSS91] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, Aug 1991.

[CM84] J. Chang and N. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, Aug 1984.

[CM95] F. Cristian and S. Mishra. The Pinwheel asynchronous atomic broadcast protocols. In *Proceedings of the Second International Symposium on Autonomous Decentralized Systems*, pages 215–221, Phoenix, AZ, 1995.

[Cri91] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[CT91] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. In *Proceedings of the 10th ACM Symposium on Principles of Distributed Computing*, pages 325–340, Aug 1991.

[DMS94] 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, Mar 1994.

[EL90] P. D. Ezhilchelvan and R. Lemos. A robust group membership algorithm for distributed real-time system. In *Proceedings of the 11th Real-Time Systems Symposium*, pages 173–179, Lake Buena Vista, Florida, Dec 1990.

[EL95] K. Echtle and M. Leu. Fault-detecting network membership protocols for unknown topologies. In F. Cristian, G. Le Lann, and T. Lunt, editors, *Dependable Computing for Critical Applications 4*, pages 69–90. Springer-Verlag, Wien, 1995.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr 1985.

[Fon94] H. Fonseca. Support environments for the modularization, implementation, and execution of communication protocols. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, Jun 1994. In Portuguese.

[GBB+95] D. Guedes, D. Bakken, N. Bhatti, M. Hiltunen, and R. Schlichting. A customized communication subsystem for FT-Linda. In *Proceedings of the 13th Brazilian Symposium on Computer Networks*, pages 319–338, May 1995.

[GT92] R. A. Golding and K. Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, University of California, Santa Cruz, May 1992.

[Hil95] M. A. Hiltunen. Membership and system diagnosis. In *Proceedings of the 14th IEEE Symposium on Reliable Distributed Systems*, pages 208–217, Bad Neuenahr, Germany, Sept 1995.

[Hil96] M. A. Hiltunen. *Configurable Distributed Fault-Tolerant Services*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, 1996. In preparation.

[HP91] N. C. Hutchinson and L. L. Peterson. The *x*-kernel: An architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, Jan 1991.

[HS93] M. A. Hiltunen and R. D. Schlichting. An approach to constructing modular fault-tolerant protocols. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 105–114, Princeton, NJ, Oct 1993.

[HS95a] M. A. Hiltunen and R. D. Schlichting. Constructing a configurable group RPC service. In *Proceedings of the 15th International Conference on Distributed Computing Systems*, pages 288–295, Vancouver, BC, Canada, May 1995.

[HS95b] M. A. Hiltunen and R. D. Schlichting. Properties of membership services. In *Proceedings of the 2nd International Symposium on Autonomous Decentralized Systems*, pages 200–207, Phoenix, AZ, Apr 1995.

[HS95c] M. A. Hiltunen and R. D. Schlichting. Understanding membership. Technical Report 95-07, Department of Computer Science, University of Arizona, Tucson, AZ, Jul 1995.

[KGR91] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In A. Avizienis and J.C. Laprie, editors, *Dependable Computing for Critical Applications*, pages 411–429. Springer-Verlag, Wien, 1991.

[KT91] M. F. Kaashoek and A. Tanenbaum. Group communication in the Amoeba distributed operating system. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS-11)*, pages 222–230, Arlington, TX, May 1991.

[Lap92] J. C. Laprie, editor. *Dependability: Basic Concepts and Terminology*. Springer-Verlag, Vienna, 1992.

[LE90] R. Lemos and P. Ezhilchelvan. Agreement on the group membership in synchronous distributed systems. In *Proceedings of the 4th International Workshop on Distributed Algorithms*, Otranto, Italy, Sep 1990.

[MAMSA94] L. Moser, Y. Amir, P. Melliar-Smith, and D. Agarwal. Extended virtual synchrony. In *Proceedings of the 14th IEEE International Conference on Distributed Computing Systems*, pages 56–65, Poznan, Poland, Jun 1994.

[MPS92] 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 2*, pages 309–331. Springer-Verlag, Wien, 1992.

[MPS93a] S. Mishra, L. L. Peterson, and R. D. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. *Distributed System Engineering*, 1:87–103, Dec 1993.

[MPS93b] S. Mishra, L. L. Peterson, and R. D. Schlichting. Experience with modularity in Consul. *Software Practice & Experience*, 23(10):1059–1075, Oct 1993.

[MSMA94] P. Melliar-Smith, L. Moser, and V. Agarwala. Processor membership in asynchronous distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 5(5):459–473, May 1994.

[OIOP93] H. Orman, E. Menze III, S. O'Malley, and L. Peterson. A fast and general implementation of Mach IPC in a network. In *Proceedings of the 3rd Usenix Mach Conference*, pages 75–88, Apr 1993.

[OP92] S. W. O'Malley and L. L. Peterson. A dynamic network architecture. *ACM Transactions on Computer Systems*, 10(2):110–143, May 1992.

[RB91] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. In *Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing*, pages 341–353, Montreal, Quebec, Canada, Aug 1991.

[RB95] R. van Renesse and K. Birman. Protocol composition in Horus. Technical Report TR95-1505, Department of Computer Science, Cornell University, Mar 1995.

[RBG+95] R. van Renesse, K. Birman, B. Glade, K. Guo, M. Hayden, T. Hickey, D. Malki, A. Vaysburd, and W. Vogels. Horus: A flexible group communications system. Technical Report TR95-1500, Department of Computer Science, Cornell University, 1995.

[Rei94] M. Reiter. A secure group membership protocol. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 176–189, May 1994.

[RFJ93] R. Rajkumar, S. Fakhouri, and F. Jahanian. Processor group membership protocols: Specification, design, and implementation. In *Proceedings of the 12th IEEE Symposium on Reliable Distributed Systems*, pages 2–11, Princeton, NJ, Oct 1993.

[RM89] B. Rajagopalan and P. McKinley. A token-based protocol for reliable, ordered multicast communication. In *Proceedings of the 8th IEEE Symposium on Reliable Distributed Systems*, pages 84–93, Seattle, WA, Oct 1989.

[SCA94] P. van der Stok, M. Claessen, and D. Alstein. A hierarchical membership protocol for synchronous distributed systems. In K. Echtle, D. Hammer, and D. Powell, editors, *Proceedings of the 1st European Dependable Computing Conference (Lecture Notes in Computer Science 852)*, pages 599–616, Berlin, Germany, Oct 1994.

[SM94] L. Sabel and K. Marzullo. Simulating fail-stop in asynchronous distributed systems. In *Proceedings of the 13th IEEE Symposium on Reliable Distributed Systems*, pages 138–147, Dana Point, CA, Oct 1994.

[SR93] A. Schiper and A. Ricciardi. Virtually-synchronous communication based on a weak failure suspector. In *Proceedings of the 23rd International Conference on Fault-Tolerant Computing*, pages 534–543, Jun 1993.

[WMK95] B. Whetten, T. Montgomery, and S. Kaplan. A high performance totally ordered multicast protocol. In K. Birman, F. Mattern, and A. Schiper, editors, *Theory and Practice in Distributed Systems (Lecture Notes in Computer Science 938)*, pages 33–57. Springer-Verlag, 1995.