An Approach to Constructing Modular Fault-Tolerant Protocols

Matti A. Hiltunen Richard D. Schlichting

TR 93-10

An Approach to Constructing Modular Fault-Tolerant Protocols[1]

Matti A. Hiltunen   Richard D. Schlichting

TR 93-10

Abstract

Modularization is a well-known technique for simplifying complex software. Here, an approach to modularizing fault-tolerant protocols such as reliable multicast and membership is described. The approach is based on implementing a protocol's individual properties as separate micro-protocols, and then combining selected micro-protocols using an event-driven software framework; a system is constructed by composing these frameworks with traditional network protocols using standard hierarchical techniques. In addition to simplifying the software, this model helps clarify the dependencies among properties of fault-tolerant protocols, and makes it possible to construct systems that are customized to the specifics of the application or underlying architecture. An example involving reliable group multicast is given, together with a description of a prototype implementation using the SR concurrent programming language. An implementation based on the *x*-kernel and RT-Mach is also underway.

March 23, 1993

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

[1]This work supported in part by the National Science Foundation under grant CCR-9003161 and the Office of Naval Research under grant N00014-91-J-1015.

1 Introduction

The *dependability* of computer systems—that is, the ability to provide correct service [Lap92]—has always been important, but is becoming more so as functions of society are increasingly computerized. Dependability is not only a concern in life-critical areas, but also in business and similar applications where incorrect system operation may mean financial loss or customer disenchantment. *Fault-tolerance* is a method for improving system dependability by introducing redundancy so that faults occurring during system operation do not result in system failure. An important type of redundancy is that found in distributed systems, where the multiple machines inherent in the architecture form a natural basis for constructing fault-tolerant systems. However, to realize this potential, the software must be constructed as *fault-tolerant software* that can continue to provide service despite failures (e.g., crashes) of the individual machines.

In this paper, we focus on the use of modularization in fault-tolerant software for distributed systems. In particular, we describe an approach to designing and implementing highly modular versions of *fault-tolerant protocols*, fundamental abstractions that have proven to be important building blocks for constructing dependable systems. Examples of such fault-tolerant protocols include reliable or atomic multicast [BSS91, CM84, CASD85, VM90], membership [Cri91a, EL90, KGR91, MPS92b] and recovery [JZ90, KT87, SY85]. This use of modularization simplifies system development and maintenance, as well as highlights some of the inherent relationships and dependencies between protocols. In addition, highly modular implementations can serve as the basis for configuring *domain-specific software* that is customized to the needs of a given application. The approach also generalizes to other types of software with fault-tolerance requirements, such as transactions and real-time protocols.

Our approach to modularization can be characterized as follows. First, the services provided by a given set of fault-tolerant protocols are analyzed to determine their constituent properties. Each such property is then implemented as a separate module called a *micro-protocol*. Next, micro-protocols that implement those properties needed by the application are composed into a *composite protocol* using a standard software framework. This framework provides an event-driven execution model, where specified micro-protocols are executed when events such as message arrival, membership change, or machine failure occur, as well as other facilities for micro-protocol interaction. Finally, a system is constructed by composing composite and simple protocols using standard hierarchical protocol composition techniques.

A high-level prototype of the software framework for composite protocols has been built using the SR concurrent programming language [AOC$^+$88]. Work is also underway on an implementation based on the *x*-kernel, a software infrastructure for composing traditional network protocols [HP91], and RT-Mach, a version of the Mach microkernel designed for execution of real-time programs [TNR90, TN91]. This research is a natural continuation of our use of modularization in Consul [Mis92, Mis91], a software platform that supports the building of fault-tolerant distributed programs structured according to the state-machine approach [Sch90].

This paper is organized as follows. In Section 2 we motivate the problem by briefly describing the more traditional Consul model of protocol composition and the problems it causes for this type of software, and then provide an overview of our approach. Section 3 describes the key part of the model: composite protocols. An example based on a suite of group oriented multicast protocols is presented in Section 4; in addition to outlining their design, we also describe the SR prototype implementation. In Section 5, we discuss configuration issues, including how dependencies and compatibility issues affect the way micro-protocols can be composed into a system. Finally, Section 5 compares our approach with similar efforts and offers some conclusions.

2 Motivation and Overview

Our approach to modularization has been motivated largely by our experience in designing and implementing Consul. To support the replicated processing implied by the state machine approach, Consul contains protocols (i.e., modules) that allow user operations to be multicast to a set of replica processes reliably and in some consistent order, to reach agreement on failures and recoveries, and to facilitate replica recovery. Figure 1 illustrates the specific protocols in the system and how they are configured using the *x*-kernel. Psync [PBS89] realizes the functionality of a partially (or causally) ordered reliable multicast, while Order transforms that into either a total or semantic-dependent order. A membership service is implemented by the combination of Failure Detection, which monitors the message flow and triggers an alarm if it suspects that another machine has failed, and Membership, which implements the agreement algorithm used to decide if a failure has indeed occurred. Recovery implements replica recovery using a combination of checkpoint and message replay.

As described more fully elsewhere [MPS92a, MPS93], part of the reason for building Consul was to test whether it was possible to build a fault-tolerant system of this type in a modular manner using traditional network protocol composition techniques. In general, we feel that this effort was successful; as can be seen from the figure, Consul is constructed from well-defined modules that each implement a single function. However, we also discovered a number of deficiencies, two of which serve as motivation for the new approach described in this paper. First, the *x*-kernel defines a simple interface between protocols that includes only operations for opening and closing connections, and for sending and receiving messages; additional operations, if needed, must be encapsulated as *control operations*. Such an interface is reasonable for traditional network protocols, which have limited interactions with one another, but overly restrictive for fault-tolerant protocols, which interact much more closely. As a result, control operations ended up being overloaded in Consul, making the system correspondingly more difficult to understand.

Second, the *x*-kernel is designed primarily to support a hierarchical composition of protocols, i.e., a protocol stack where one high-level protocol depends on one low-level protocol. In Consul, however, several protocols "at the same level" must cooperate to implement a set of services. For example, Membership, Order, and Failure Detection all cooperate with each other, but without being stacked on top of one another. To find a way around this limitation in the *x*-kernel model, we had to implement two additional protocols (Re)Start and Divider, whose only function is to coordinate the parallel composition of protocols. Psync also had to serve a function in this regard by reflecting messages originating within protocols and destined for the network up to the other protocols at the same level.

Based on these observations, we have developed a new model for protocol composition that is more oriented towards the needs of fault-tolerant systems. For example, in addition to traditional hierarchical composition, our model adds parallel composition in which micro-protocols can be combined using a software framework to yield a composite protocol. Such a protocol presents an external interface indistinguishable from a traditional simple protocol, which allows it to be composed hierarchically with other composite and simple protocols using the standard *x*-kernel model. Execution of micro-protocols within a framework is based on an event-driven model, which alleviates the need for control operations to support interaction.

Figure 2 illustrates the overall organization of a Consul-like system structured in this way. The composite protocol is indicated by the double lines, with the names of its constituent micro-protocols listed inside. Note the differences between this and Figure 1. For example, the extra protocols (Re)Start and Divider are no longer needed since the parallel composition aspect of the framework provides the necessary functionality. Also, note that we have been able to modularize Psync further,

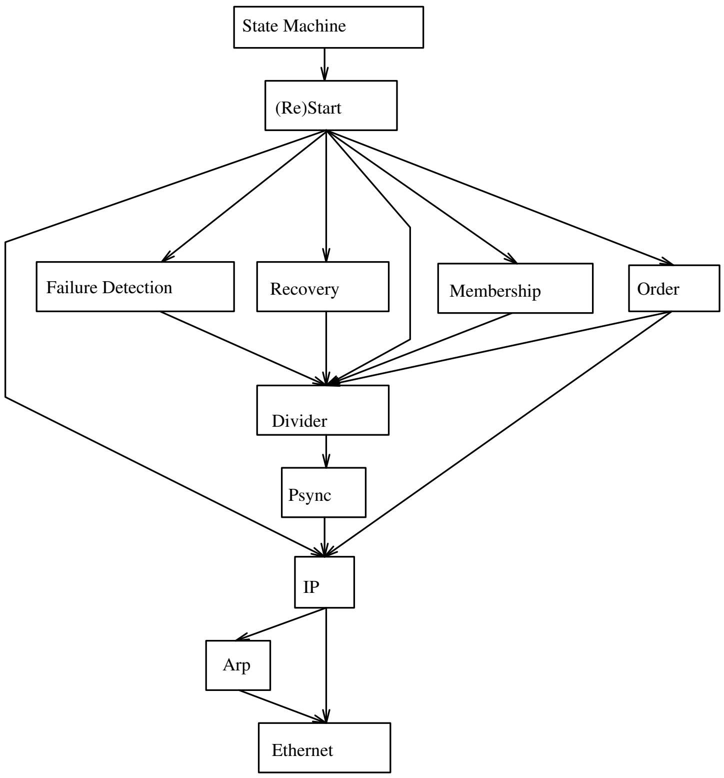

Figure 1: Consul configuration graph

using reliability, causality, and stability[1] to realize the equivalent functionality. Provisions for more complicated interactions among the protocols are provided by the shared variable and event mechanisms within the framework.

Finally, it is important to emphasize that, although our original motivation was Consul, this two-level model does far more than address Consul-specific problems. For one thing, the types of problems we encountered are those that would be encountered in attempting to design and implement *any* fault-tolerant system in a modular way. Also, the emphasis on supporting a finer granularity of modularization efficiently, as illustrated by the way in which Psync has been implemented as a collection of micro-protocols, goes significantly beyond Consul. Such a capability is a necessary prerequisite for, among other things, further experimentation with domain-specific software. Finally, unlike Consul, this model is intended to apply to fault-tolerant protocols other than those associated with the state machine approach.

3 Composite Protocols

A composite protocol presents the external interface of a simple protocol, but is constructed by the parallel composition of micro-protocols executed in an event-driven manner. Thus, the important components of a composite protocol are as follows:

[1] A message is *stable* when it is followed in the causality graph by a message from every other replica.

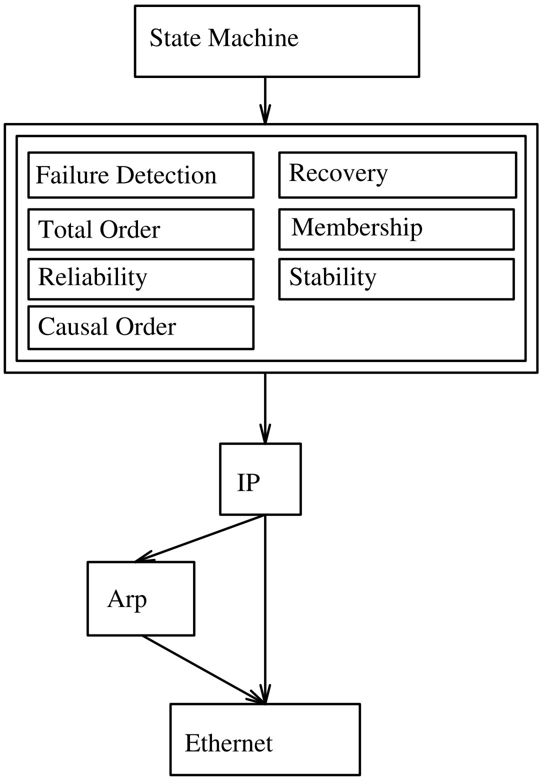

Figure 2: Equivalent configuration graph using new model

- *Micro-protocols*: A section of code that implements a single well-defined property.
- *Events*: An occurrence that causes one or more micro-protocols to be invoked. Some events of interest are predefined (e.g., message arrival); others are defined by micro-protocols (e.g., change in group membership).
- *Framework*: A software infrastructure that implements the event registration and triggering mechanism, and contains shared data (e.g., messages) that can be accessed by more than one micro-protocol.

This model is depicted in Figure 3. In the middle is the framework, which contains a shared data structure—in this case a bag of 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. We now elaborate further on these three aspects.

3.1 Micro-protocols

A micro-protocol is structured as a collection of *event handlers*, which are procedures that are invoked when a particular event is detected and triggered. Specifically, a micro-protocol is defined by the following:

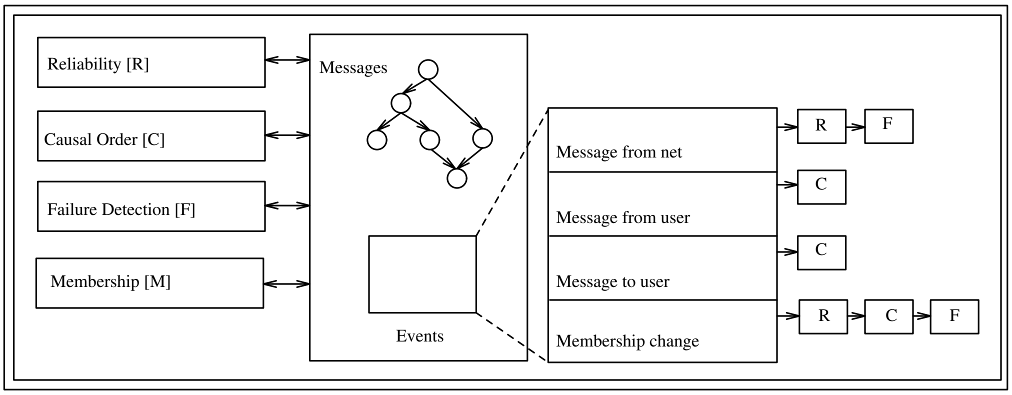

Figure 3: A composite protocol

- *Header*: Contains import/export information, such as events that are defined by this micro-protocol, events that are triggered by this micro-protocol, shared data updated by this micro-protocol, and shared data read by this protocol.
- *Local data*: Data accessible only from within the micro-protocol. This data is persistent, i.e., the variables keep their assigned values as long as the framework is active.
- *Initialization*: Initializes local data, defines new events, and registers handlers for events.
- *Event handlers*: Consists of a header and a procedure. The header defines the names and argument types similar to the way a normal procedure or function is declared.

This way of building protocols may seem difficult at first, but actually fits very well with the way people typically think about protocols. For example, one often sees protocols described as a collection of rules of the following form:

when $< condition >$ do $< action >$

where $< condition >$ may be message arrival, timeout, or a more complex condition, e.g., deadlock. In our model, these rules map easily to event-handlers.

Often, there is more than one way to structure a micro-protocol as a collection of event-triggered actions. In fact, there often may not even be a unique set of events on which a micro-protocol depends. The choice of the best solution is generally a tradeoff between performance and the ease of reading and understanding the behavior of the micro-protocol. From a performance point of view, one of the most important issues is the cost of implementing event detection. If the event is simple, such as message arrival, or the event can only be triggered by a single micro-protocol, such as making some property of a message true, the implementation will be very efficient. On the other hand, it is also possible to design more complex events, such as "The message with identifier I has arrived or the sender of the message has failed." In these cases, event detection is not easy to do efficiently. We are currently working on implementation techniques for handling such complex events in a reasonable manner.

The issue of the best way to divide fault-tolerant protocols and systems into micro-protocols is addressed in Section 4 in the context of a group oriented multicast example.

3.2 Events

An event definition contains a symbolic name, an interface specification for the handlers that field events with this name, and the *event type*. The event type is defined along two dimensions: *parallelism semantics* and *activation semantics*. Parallelism semantics are defined as either parallel or serial. A parallel event is one in which the event handlers associated with the event are activated (logically) in parallel, while a serial event is one in which the handlers are activated in sequence. Parallel activation is necessary if a handler may delay during its execution; serial activation is necessary if the handlers have ordering dependencies. Activation semantics are defined as either synchronous or asynchronous. Synchronous event activation is one in which the caller—that is, the micro-protocol triggering the event—waits until all the invoked micro-protocols complete, while an asynchronous is one where the caller continues without waiting for completion.

The abstract syntax for the operation provided by the framework for defining new events is as follows:

define(NAME,*handler_type_definition*,PAR/SER,SYNC/ASYNC)

Here, NAME is the symbolic name, $handler\_type\_definition$ is the interface specification, and the final two arguments specify the parallelism and activation semantics, respectively. Some events, described below, are predefined implicitly within the framework based on their nearly universal usage in applications of this type.

Handlers are associated with events when a micro-protocol registers a handler for a named event, as follows:

register(NAME, *handler*)

Events are detected and associated handlers triggered by the framework or micro-protocols. The framework can only trigger predefined events. Micro-protocols, on the other hand, can trigger either defined or predefined events. Based on our experience, the micro-protocol that defines a given event is usually the one that triggers it, but other uses are not precluded by the model. The following notation is used to trigger events within micro-protocols:

trigger(NAME, *arguments*)

Provisions are also made for events that are triggered by the passage of time. For example, a micro-protocol can ask to be notified at an absolute time T_1, after T_2 time units have elapsed, or every T_3 time units. These facilities are especially useful for constructing *clock-driven* systems in which many of the algorithms are based on executing certain actions at specified times [Ver90].

3.3 Framework

The framework provides an infrastructure for composing micro-protocols in parallel to form a composite protocol. The two major components of this infrastructure are provisions for shared data and the event mechanism. The latter includes the *define*, *register*, and *trigger* operations defined above, as well as the means for detecting predefined events such as message arrival. The framework also implements the event-handler execution mechanism using threads provided by the underlying system; in the SR prototype, these threads are implemented by the language runtime system, while in the final version they are expected to be provided by the RT-Mach microkernel.

To define a framework, then, two pieces of information are specified:

- *Set of predefined events*: The event name and the interface specification of associated event handlers.
- *Shared data*: Declaration of shared data structures together with access rights.

The access rights specify which micro-protocols can read and/or update the data. Our experience is that each piece of shared data is updated by only a single process, but potentially read by many.

3.4 Discussion

This model, and especially the provisions for parallel composition of micro-protocols into composite protocols, has both conceptual and practical advantages. Conceptually, the model helps in determining the fundamental properties of a given fault-tolerant protocol and highlights their dependencies. By helping to improve our understanding of such issues, the model facilitates the construction of more dependable systems.

From a practical point of view, the model facilitates customization and reuse. Customization is easier than a corresponding monolithic system since individual properties can be implemented as separate micro-protocols. As a result, the system builder can choose a configuration that contains exactly those properties required for a given application, rather than being forced to include properties that are either unnecessary (e.g., consistent message order when none is needed) or too strong (e.g., consistent total order when causal order is sufficient). Moreover, the system can also be customized for the particular hardware being used simply by replacing a particular micro-protocol with another that implements the same property but uses a different algorithm. This feature also makes it easy to experiment with different algorithms for realizing properties, or to reuse micro-protocols in subsequent systems.

Finally, another practical advantage is that the model simplifies the problem of dealing with dependences and interactions. In existing hierarchical models such as that used for Consul, non-standard dependencies and interactions between modules had to be dealt with in an ad hoc fashion. The net result was an implementation where the physical structure of the software did not really reflect the underlying logical structure. In this model, however, the flexibility of the event mechanism makes it much easier to reflect this logical structure in the software. This feature simplifies the organization and makes it more understandable.

4 Using the Model: Group Oriented Multicasts

As mentioned, this model has been used to construct a small system containing micro-protocols for group oriented multicast and related services. Here, we describe this example using an abstract syntax similar to what will be used in the final system. The specifics of the SR prototype are addressed in Section 4.3.

4.1 Multicast Properties

When surveying existing multicast systems, prototypes, and algorithms, it is clear that there is a large amount of variance in the specific properties that they provide to the user. These range from the V-system [CZ85], which provides only unreliable multicast, to a number of more recent systems, like Consul, ISIS [BSS91], and Transis [ADKM92], which provide reliable multicasts with a number of different ordering properties. Our approach to constructing these protocols, then, is based on identifying these properties and implementing each as a micro-protocol.

Of the properties found in these systems, we have identified three that seem especially important from the user's perspective:

- *Reliability*: A message is delivered to all processes or a subset thereof despite transmission failures.
- *Order*: Messages are delivered in some consistent order to all processes in the group.
- *Atomicity*: A message is delivered either to all group members or to no group member.

There are also other fundamental services, like membership, that are used within the system to implement these properties.

Each of these properties actually has many variants depending on the specifics of the definition. For example, Reliability could range from one guaranteeing that at least a certain number of the receivers receive the message (k-quorum), to one that guarantees that every receiver, including those temporarily down, will eventually receive the message. The different definitions take into account, among other things, the particular failure model assumed and the semantics of the group (e.g., if temporarily down sites are considered members of the group or not).

For Order, properties range from simple FIFO to causal, total, and different application-specific orderings. Similar to Reliability, slightly different guarantees result when different failure models and group semantics are taken in account. This is especially true if ordering is guaranteed across group partitions.

Atomicity can have different definitions based on whether there are time bounds or not. For example, there may be a bound by which the decision to deliver or not deliver the message must be made, versus "eventual" atomicity (i.e., if one receiver receives the message, eventually everybody will receive the message). The definition of atomicity also depends heavily on the semantics of the group, especially whether failed processes are considered still to be group members or not.

4.2 Framework

In this example application, the shared data supported by the framework is a bag of messages, each of which has associated *attribute fields*. These fields are used, for example, to maintain the edges of a causality graph and to indicate which properties are to be enforced on the message. As far as events are concerned, the following are predefined by the framework:

- *REC_NET*: Arrival of a message from a protocol below this one (i.e., from the direction of the network).
- *REC_USER*: Arrival of a message from a protocol above this one (i.e., from the direction of the user application).
- *SEND_NET*: Message about to be sent towards the network. Enables micro-protocols to update fields in the message header and do local bookkeeping.
- *SEND_NET_DONE*: Message has been sent towards the network. Used, for example, by a reliability protocol based on positive acknowledgments to ensure that every member acknowledges the message.

- *SEND_USER*: Message about to be sent towards the user. Enables micro-protocols that enforce certain message properties to verify that the message can leave the framework. The actual message delivery occurs only after all relevant micro-protocols have performed such a check.
- *SEND_USER_DONE*: Message has been sent towards the user. Mostly used by micro-protocols to do bookkeeping.

All other events, like membership changes, are defined and triggered by micro-protocols.

4.3 Micro-protocols

Our current suite of micro-protocols are based on the assumptions that the network is asynchronous and that machines experience only *fail-silent* (or crash) failures where the processor halts and all of volatile storage is lost [PSB+88]. Moreover, we assume that a failed process is removed from the group membership while it is down, but may join the group again after recovery.

The following are the specific micro-protocols that are used to realize various forms of reliable multicast and associated services.

- *FIFO*: Implements FIFO message ordering for messages sent by any one sender. There are versions that implement both unreliable and reliable transmission. The unreliable version guarantees that a given message will be delivered in FIFO order, but with the possibility that the sequence may be incomplete.
- *Causality*: Implements consistent causal ordering of multicast messages. There are versions that implement both unreliable and reliable transmission. Similar to unreliable FIFO, unreliable causal order guarantees that causal order is maintained, but with the possibility that the collection of messages may be incomplete.
- *Total*: Implements consistent total ordering of multicast messages. Several version using different algorithms are employed.
- *Reliability*: Ensures that every group member receives every message despite transmission failures. There are two versions: one based on a negative acknowledgment scheme and the other on a positive acknowledgment scheme.
- *Stability*: Ensures that a message will not be delivered upwards before becoming stable. Such a a message is one that is followed in the causality graph by a message from every other site. This property can be viewed as a form of atomicity.
- *Membership*: Implements the agreement aspect of a membership service. Separate micro-protocols deal with process failures and joins. Triggers an event indicating membership change.
- *Monitor*: Warns of a possible machine failure if no message is received from a group member within T_0 seconds.
- *Liveness*: Ensures that a machine sends a message every T_1 seconds; sends a null message if no application messages have been sent.
- *Validity*: Verifies that the sender of a message received from below is a valid member of the group.

Several additional micro-protocols are used for debugging purposes.

As examples, a reliability micro-protocol using negative acknowledgements and a monitor micro-protocol are outlined in Figures 4 and 5, respectively. It should be noted that these are much simplified versions of the actual micro-protocols. For instance, in the reliability micro-protocol, a retransmission request is sent only once per missing message, whereas in reality, such requests would continue until either the message is received or the membership protocol informs us that every machine that received the message has failed.

```
micro-protocol Reliability
    event-handler failure_handler(site_id: int)
        . . . update local membership list . . .
    end

    event-handler recovery_handler(site_id: int)
        . . . update local membership list . . .
    end

    event-handler send_reliability(np: ptr node)
        . . . update message header and local state . . .
    end

    event-handler receive_reliability(mp: ptr message)
        if "message is a retransmission request" then
            . . . retransmit message if it is in the local bag . . .
        else
            for "each predecessor of the message in the graph " do
                if "predecessor not in the bag of messages" then
                    . . . send retransmission request . . .
                end if
            end for
        end if
    end

    register(SEND_NET,send_reliability)              # initialization code
    register(REC_NET,receive_reliability)
    register(SITE_FAILURE,failure_handler)
    register(SITE_RECOVERY,recovery_handler)
end Reliability
```

Figure 4: Outline of Reliability micro-protocol

4.4 SR Prototype Implementation

Prototype versions of the above framework and micro-protocols have been implemented using the SR concurrent programming language [AOC+88]. In the prototype, each machine hosting members of the multicast group is implemented as an SR *virtual machine*. A framework is then implemented as an SR *resource*, an object that contains local variables, procedures, and processes, and exports operations for use by other resources. A framework resource exports operations to the frameworks above and

```
micro-protocol Monitor
    var message_from[1:"max number of sites"]: bool

    event-handler monitor_failure_handler(site_id: int)
        . . . update local membership list . . .
    end

    event-handler monitor_recovery_handler(site_id: int)
        . . . update local membership list . . .
    end

    event-handler monitor_reception(mp: ptr message)
        message_from["sender of msg"] := true
    end

    event-handler monitor_check_failure()
        for "each site in the current membership" do
            if (not message_from["site"]) then
                trigger(SUSPECT_FAILURE,"site")
            end if
            message_from["site"] := false
        end for
        register_timeout(T0,monitor_check_failure)
    end

    define(SUSPECT_FAILURE,(site_id: int),PAR,ASYNC)
    register(REC_NET,monitor_reception)
    register(SITE_FAILURE,monitor_failure_handler)
    register(SITE_RECOVERY,monitor_recovery_handler)
    register_timeout(T0,monitor_check_failure)
end Monitor
```

Figure 5: Outline of Monitor micro-protocol

below for use in transmitting messages up and down the protocol stack. In our case, each simulated machine consists of three frameworks; in addition to the one implementing our group oriented multicast micro-protocols, there are frameworks that simulate the network and the application, respectively. The latter two are degenerate frameworks and do not implement full event handling.

In the prototype, an event handler is simply a regular SR procedure. Association of an event handler with a named event is done by invoking a "register" routine in the framework and passing a capability (i.e., pointer) to the procedure. The framework maintains a table containing this event/handler association. Event handlers are triggered using the normal SR invocation mechanism. At this point, no provisions have been made for defining or using the micro-protocol header information described in Section 3.1.

Message transmission failures and machine failures can only be simulated in the prototype. The network framework simulates message transmission failures. Specifically, a multicast to N receivers is implemented by N point-to-point messages, so the program uses a probability distribution to decide if a particular point-to-point message will be transmitted or not. Simulating transmission failures has turned out to be very useful since with very large failure probabilities (0.2–0.5/transmission), short simulations are often enough to bring up interesting failure scenarios. Machine failures are simulated by having the application framework set a variable that is shared among all resources on a virtual machine. When this occurs, the network and multicast frameworks cease forwarding messages, and are terminated along with the application framework. To simulate recovery, the frameworks are recreated following some specified time interval.

5 Configuration Issues

Our model attempts to simplify the interactions between micro-protocols. However, while configuring a set of micro-protocols into a system, the designer still has to understand *dependencies* between the micro-protocols to be sure that the combination works correctly and provides the service expected. Here, a micro-protocol MP_1 is said to depend on another micro-protocol MP_2 if the correctness of MP_1 relies on the correctness of MP_2 [Cri91b]. Direct interactions between micro-protocols, such as caused by triggering and fielding events, define explicit dependencies between micro-protocols. Unfortunately, all dependencies are not as explicit; often there are other compatibility issues that affect how and when micro-protocols can be combined. In the following, we discuss these two configuration issues.

5.1 Direct Interactions

If two micro-protocols interact in any sense—that is, if one of them in any way affects the behavior of the other—the interaction defines a dependency between the two protocols. For example, in a system consisting of simple and composite protocols composed hierarchically, one protocol using the service interface of another defines an interaction. This specific interaction and the dependency it creates is no different from those in systems like Consul based on hierarchical composition, so here we concentrate on the interactions between micro-protocols within a single framework, i.e., between micro-protocols that implement one composite protocol.

One form of interaction is when a micro-protocol triggers an event that is fielded by another micro-protocol. The dependency created by this interaction can be in either direction, that is, from the triggering to the fielding micro-protocol or the other way around. An example of the first type in our example system is caused by the event signaling a group membership change, which is raised by the membership micro-protocol and fielded by various other micro-protocols. The other type occurs when a protocol triggers an event to request a service.[2] An example of this type of dependency can be found again in the membership micro-protocol, which triggers the event REC_USER to send a causally ordered reliable multicast message to all the other group members.

Data sharing is another form of interaction. In this case, the dependency is between the micro-protocol that updates a shared variable or field in a shared data structure and the micro-protocols that read the data. In other words, the readers depend on the writers. Each shared data structure usually has one writer and one or more readers, a characteristic that simplifies this issue, as well as the handling of concurrent access. An example of this type of dependency is a total ordering protocol that uses the causality graph built by the causality protocol to implement total order.

Message exchange between two different micro-protocols is the third form of interaction that we have observed. If a micro-protocol, say MP_1, expects to receive a message sent by some other micro-protocol, say MP_2, then MP_1 depends on MP_2. An example of this type of dependency in our example

[2] This is, in fact, very close to a "uses" relationship.

is between the Liveness micro-protocol, which ensures that a message is sent every T_1 seconds, and the Monitor micro-protocol, which checks that a message is received from each machine every T_2 seconds.

Our experience is that it is usually easy to detect interactions between micro-protocols, and therefore, to find these dependencies. Although our prototype implementation does not do so, it would even be possible to automatically generate dependency information if micro-protocol headers were complete enough.

5.1.1 Dependency graphs

Dependencies based on interactions can be represented as a *dependency graph*, where the edges are micro-protocols and a directed edge from MP_1 to MP_2 indicates that MP_1 depends on MP_2. Figure 6 illustrates some of the dependencies between micro-protocols from the multicast example given in Section 4. In this graph, the solid lines represent absolute dependencies, in the sense that a micro-protocol would not operate correctly without the micro-protocols upon which it depends. The dashed lines, on the other hand, represent dependencies that are present in our prototype, but which can be viewed as optimizations. All dashed lines here are created by the membership micro-protocol notifying the other micro-protocols about membership changes. For example, the FIFO and causal ordering micro-protocol use this information to stop waiting for messages that will never arrive due to a machine failure.

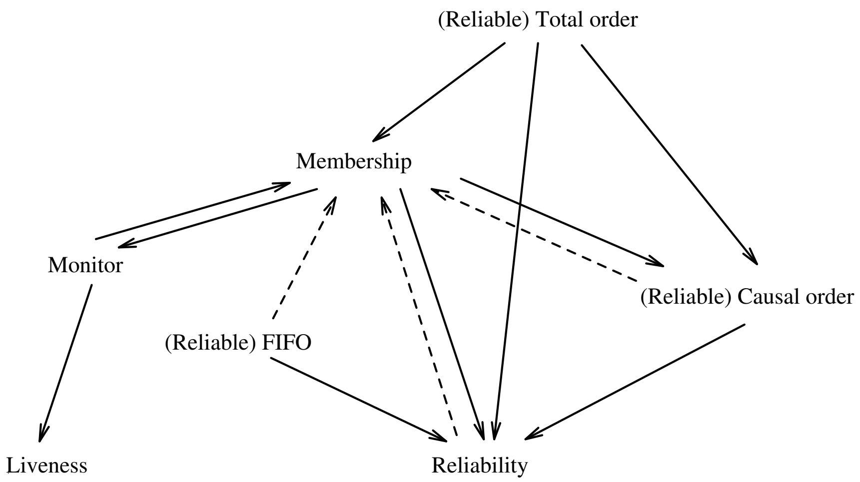

Figure 6: Dependency graph

It is also important to note that these dependencies do not represent inherent dependencies between the properties implemented by micro-protocols, but rather just dependencies resulting from our particular realization. For example, our membership micro-protocol uses causally ordered reliable multicasts for reaching agreement [MPS92b], and so depends on the micro-protocols implementing those properties. In contrast, membership services such as [KGR91] depend instead on totally ordered reliable multicasts.

The net result of these dependencies is that configuring a system is not as simple as just selecting micro-protocols. A system is built by first choosing those micro-protocols that provide the required properties, and then choosing all the micro-protocols in the transitive closure of the selected micro-protocols in the dependency graph. For example, in the dependency graph of Figure 6, if we choose to use Total order we have to include the micro-protocols for Membership, Monitor, Liveness, Causal order, and Reliability. However, if the application only requires reliable causally ordered multicasts, only the micro-protocols for Reliability and Causal order must be included.

5.2 Micro-Protocol Compatibility

Micro-protocol interactions and the dependencies they create are not the only thing that affects how and when micro-protocols can be configured into a system. Another is whether the micro-protocols are *compatible*, that is, based on the same set of assumptions regarding such things as time, group semantics, and the underlying architecture. The need for compatibility implies, among other things, that the way in which a micro-protocol implements a certain property can depend on what other micro-protocols it will be used with and how those properties are implemented.

As an example, consider a micro-protocol that implements some kind of consistent message ordering in a multicast system. It turns out that the way in which this property is implemented differs depending on whether or not there is also a micro-protocol implementing reliable message transmission configured into the system. To see this, consider first an Order micro-protocol that does not rely on reliable transmission, but rather is defined to just deliver received messages in the required order. In this case, Order can simply drop messages that arrive too late—that is, after a subsequent message in the ordering has already been delivered—since no reliability guarantee is provided. Note, however, that combining this particular variant of Order with one that *does* guarantee reliability will not result in the desired semantics since Order may still drop messages if they arrive out-of-order. Conversely, consider an ordering protocol that does exploit message reliability. In this case, an out-of-order message is retained until all its predecessors in the ordering have also arrived. Running this particular ordering micro-protocol without reliability, however, will deadlock if a message does in fact become lost. What this example shows, then, is that reliability and ordering must be built using the same set of assumptions to function correctly together.

Similar compatibility issues abound. An example where the compatibility assumption involves time is when a monitor protocol must rely on a liveness protocol to guarantee that a message is sent every T seconds. Similarly, if stability is defined to include group members that are temporarily down, a stability micro-protocol must rely on a recovery micro-protocol to eventually restore activity on failed machines. In this case, stability also assumes that a message from every machine will eventually be received, which implies that it is relying on the reliability micro-protocol as well.

6 Conclusions

Modularization is a well-known technique for simplifying the development of complex software. Here, we have described an approach to modularizing fault-tolerant protocols that attempts to gain such benefits for this type of software. Our approach is based on constructing composite protocols from a collection of micro-protocols, and then combining composite and simple protocols into a system using standard hierarchical techniques. A software framework supporting shared data and event-driven execution is used to compose micro-protocols, each of which is designed to implement a single, well-defined property. In addition to clarifying dependencies among properties, such an approach has practical benefits as well. For example, it makes it easier to construct different variants of fault-tolerant protocols, and to configure a system in which the particular protocols are tuned to the environment and characteristics of the application.

Other projects in the area of fault-tolerant computing have also explored use of modularization. The ANSA system [OOW91] allows the user to customize the abstractions provided by the system to a certain extent; for example, it provides *selective group transparencies*, which allow certain system modules to be replaced by user-written modules. In [Gol92], an approach to building group communication systems is described. The approach is based on having four fixed components—the application, message delivery, message ordering, and group membership—that communicate through three shared data structures—a message log, message summary information structure, and group view. While representing a modular structure, the components and data structures are fixed, which means that the interactions between the components are also fixed. Our model, on the other hand, is more general since it offers a general event mechanism, general shared data structures, and configuration of any number of components (micro-protocols). Finally, the goals put forth in [Bla91] in the area of transaction processing are related to ours in the sense that it attempts to identify similar orthogonal properties, but for transactions rather than fault-tolerant protocols.

Future work will concentrate along several lines of investigation. One is continuing our work on implementing a version of the software framework based on the *x*-kernel and RT-Mach. Such a framework will provide a more realistic environment for constructing fault-tolerant protocols and experimenting with domain-specific software. Another is exploring issues related to configuration policy and its supporting mechanisms. For example, allowing the software to dynamically reconfigure itself at runtime may be expensive to implement, but could open the door to easy implementations of certain hybrid or adaptive approachs that have proven useful in this context. Finally, we will continue to investigate dependency issues since the classification in Section 5, while useful, is far from complete. By doing do, we hope to improve our understanding and facilitate the development of configurable modules that can be used in a variety of fault-tolerant applications.

References

[ADKM92] 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, July 1992.

[AOC+88] Gregory R. Andrews, Ronald A. Olsson, Michael A. Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin, and Gregg Townsend. An overview of the SR language and implementation. *ACM Transactions on Programming Languages and Systems*, 10(1):51–86, January 1988.

[Bla91] A. Black. Understanding transations in an operating system context. *ACM Operating Systems Review*, 20(1):73–76, Jan 1991.

[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.

[CASD85] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. In *Proceedings of the Fifteenth International Symposium on Fault-Tolerant Computing*, pages 200–206, Ann Arbor, MI, Jun 1985.

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

[Cri91a] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[Cri91b] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.

[CZ85] D. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 2(3):77–107, May 1985.

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

[Gol92] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, Dept of Computer Science, University of California, Santa Cruz, Santa Cruz, CA, 1992.

[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.

[JZ90] D. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, pages 462–491, 1990.

[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.

[KT87] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *IEEE Transactions on Software Engineering*, SE-13(1):23–31, Jan 1987.

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

[Mis91] S. Mishra. *Consul: A Communication Substrate for Fault-Tolerant Distributed Programs*. PhD thesis, Dept of Computer Science, University of Arizona, Tucson, AZ, 1991.

[Mis92] S. Mishra. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report 92-06, Dept of Computer Science, University of Arizona, Tucson, AZ, 1992.

[MPS92a] S. Mishra, L. L. Peterson, and R. D. Schlichting. Experience with modularity in Consul. Technical Report 92-25, Dept of Computer Science, University of Arizona, Tucson, AZ, 1992.

[MPS92b] 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, Vienna, 1992.

[MPS93] S. Mishra, L. L. Peterson, and R. D. Schlichting. Modularity in the design and implementation of Consul. In *Proceedings of the First IEEE Symposium on Autonomous Decentralized Systems*, Kawasaki, Japan, March 1993. To appear.

[OOW91] M. Olsen, E. Oskiewicz, and J. Warne. A model for interface groups. In *Proceedings of the Tenth Symposium on Reliable Distributed Systems*, pages 98–107, Pisa, Italy, Sep 1991.

[PBS89] L. L. Peterson, N. C. Buchholz, and R. D. Schlichting. Preserving and using context information in interprocess communication. *ACM Transactions on Computer Systems*, 7(3):217–246, Aug 1989.

[PSB+88] D. Powell, D. Seaton, G. Bonn, P. Verissimo, and F. Waeselynk. The Delta-4 approach to dependability in open distributed computing systems. In *Proceedings of the Eighteenth Symposium on Fault-Tolerant Computing*, Tokyo, Jun 1988.

[Sch90] F. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec 1990.

[SY85] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transactions on Computer Systems*, 3(3):204–226, Aug 1985.

[TN91] H. Tokuda and T. Nakajima. Evaluation of real-time synchronization in Real-Time Mach. In *Proceedings of the USENIX 1991 Mach Workshop*, Oct 1991.

[TNR90] H. Tokuda, T. Nakajima, and P. Rao. Real-Time Mach: Towards predictable real-time systems. In *Proceedings of the USENIX 1990 Mach Workshop*, Oct 1990.

[Ver90] P. Verrissimo. Real-time data management with clockless reliable broadcast protocols. In *Proceedings of the Workshop on Management of Replicated Data*, pages 20–24, Houston, TX, Nov 1990.

[VM90] P. Verissimo and J. Marques. Reliable broadcast for fault-tolerance on local computer networks. In *Proceedings of the Ninth Symposium on Reliable Distributed Systems*, pages 54–63, Huntsville, AL, Oct 1990.