Experience with Modularity in Consul

Shivakant Mishra,[1] Larry L. Peterson, and Richard D. Schlichting

TR 92-25

Abstract

The use of modularity in the design and implementation of complex software simplifies the development process, as well as facilitating the construction of customized configurations. This paper describes our experience using modularity in Consul, a communication substrate used for constructing fault-tolerant, distributed programs. First, Consul is presented as an example of how modularity is feasible in both the design and the implementation of such systems. Second, modularity issues that arose during development are discussed. These include deciding how the system is divided into various modules, dealing with problems that result when protocols are combined, and ensuring that the underlying object infrastructure provides adequate support. The key observation is that dependencies between modules—both direct dependencies caused by one module explicitly using another's operation and indirect dependencies where one module is affected by another without direct interaction—make modularization especially difficult in systems of this type. While our observations are based on designing and implementing Consul, the lessons are applicable to any fault-tolerant, distributed system.

August 28, 1992

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

[1] Current address: Dept. of Computer Science and Engineering, University of California San Diego, La Jolla, CA 92093, USA

INTRODUCTION

A common way of simplifying the development of large software systems is through the use of modularity. With this approach, the individual functions of a system are designed and implemented independently of one another as fully-encapsulated modules, with interaction between modules taking place only through well-defined interfaces. Dividing the software in this way and controlling its interactions has a number of advantages. One is that it is easier to design, implement, debug, and optimize individual modules than it is an entire monolithic system. Another is that it forces the builder to develop a better understanding of the interactions and dependencies among modules, leading to a more dependable system with better internal structure. A third is that it facilitates the development of customized systems for specific applications, since only those modules implementing functions essential to the specific needs of the application need be included.

Despite these advantages, the use of modularity has lagged in the development of software for *fault-tolerant, distributed systems*, or at least has not been used to the point where it is emphasized and acknowledged explicitly in the literature. As its name implies, a fault-tolerant, distributed system has two fundamental characteristics. First, it is built on a distributed architecture in which multiple machines are connected by a network without the benefit of shared memory. Second, the system must be able to continue functioning despite failures such as machine crashes. These properties make the software for such systems inherently complex, largely due to the need to deal with the asynchrony associated with multiple machines and network transmission, as well as the uncertainty caused by network and machine failures. Given that modularity is fundamentally a technique for controlling complexity, its application to software of this type has the potential to reap large benefits.

This paper describe our experience using modularity in the design and implementation of a communication substrate called Consul [Mis91, MPS91a] that supports the construction of fault-tolerant, distributed software structured using the *state machine approach* [Sch90]. Consul provides various *fault-tolerant services* such as group-oriented multicast, membership, and recovery, which simplify the problems associated with consistently ordering events and dealing with failures. These services are realized using *protocols* as the fundamental modules of the system. These protocols are, in turn, implemented in an underlying *object infrastructure*, in our case, the *x*-kernel [HP91].

Our focus on modularity in Consul takes two forms. First, we present Consul as an example of how modularity can actually be achieved in a system of this type. Unlike similar systems, in our approach, each fault-tolerant service is designed and implemented independently of the others as one or two protocols, with a system then being constructed from a library of such protocols. Second, we outline some of the issues that arose as a result of our emphasis on modularity, and describe how these were dealt with in the context of Consul. These include deciding how the system is divided into various modules and designing their interfaces, dealing with correctness and efficiency problems that result when protocols are combined, and ensuring that the underlying object infrastructure provides adequate support.

A key theme that unifies many of the issues we discuss is the problem of dependencies between modules. In systems such as Consul, a given fault-tolerant service relies on other services to realize its functionality, and hence, *depends* on their correctness to ensure its own [Cri91]. These dependencies are manifested in the implementation either directly, as when one protocol invokes an operation on another, or indirectly, as when one protocol relies on another to establish a certain state. While both types of dependencies complicate the process of developing a correct and efficient system, the latter are most difficult to handle since they can be viewed as akin to side-effects. One of the main contributions of this paper is to highlight these direct and indirect dependencies, many of which are inherent in fault-tolerant, distributed system and not specific to Consul.

The following two sections describe modularity in Consul at two different levels. The first defines the set of abstract services supported by Consul and outlines how these services are mapped onto a set of protocol modules; we refer to this as *design modularity*. The second describes how Consul's protocol modules are realized in a particular object infrastructure; we refer to this as *implementation modularity*. Subsequently, issues and problems relating to these two levels of modularity are discussed. The final section contains conclusions.

DESIGN MODULARITY

From the application's perspective, Consul provides a collection of fault-tolerant services that collectively support the state machine model of distributed computing [Sch90]. In this approach, the application maintains *state variables* that are modified in response to *commands* received from other state machines. Execution of a command is deterministic and atomic with respect to other commands. The output of a state machine, that is, the sequence of commands to other state machines or the environment, is completely determined by the sequence of input commands.

A fault-tolerant version of a state machine is implemented by replicating the state machine and running each replica in parallel on a different processor in a distributed system. Key requirements for implementing the state machine approach include maintaining replica consistency at all times and integrating repaired replicas following failure. The fault-tolerant services found in Consul are designed specifically to support these requirements. For example, the *multicast* service provides atomic (i.e., all or nothing) message delivery and a consistent ordering among all recipients, which makes it ideal for disseminating commands to state machine replicas.

Figure 1 illustrates the abstract fault-tolerant services found in Consul and the dependencies among them. In this figure, the rectangles are services, with an arrow from service S_1 to service S_2 indicating that the correctness of S_1 depends on the correctness of S_2 [Cri91]; the edge labels indicate the property that induces the dependency. At the top is the state machine that represents the application program; it depends directly on two services: multicast and recovery. As already mentioned, multicast is a communication service that allows a message to be transmitted asynchronously to a group of processes atomically and in a consistent order, while recovery deals with restoring the state of a failed state-machine replica upon restart. Membership provides a consistent view of which processors are functioning and which have failed at any given moment in time. Membership is used by the recovery service when a replica recovers and the multicast service to implement a consistent total order; it also depends on multicast to disseminate messages to instances of the membership service on other machines. The time service provides the abstraction of a common time base on all the machines in a distributed system despite the lack of a single physical clock. In Consul, this service is realized using *logical clocks* [Lam78], and is used by multicast to consistently order messages. Finally, we note that, while this division of functionality into abstract services is somewhat arbitrary, a large number of systems use these services or variants thereof [BJ87, BSS91, CDD90, KM85, KDK$^+$89, PSB$^+$88]. Further discussion of these services, their interrelationships, and the systems that use them can be found in [MS92].

We now turn our attention from the abstract services provided by Consul to the set of protocol modules that realize these services in the substrate. A copy of these protocols resides on each machine in a distributed system, and provides an interface between the application program in the form of the state machine replicas and the underlying network. The communication network is assumed to be asynchronous, with no bound on the transmission delay for a message between any two machines.

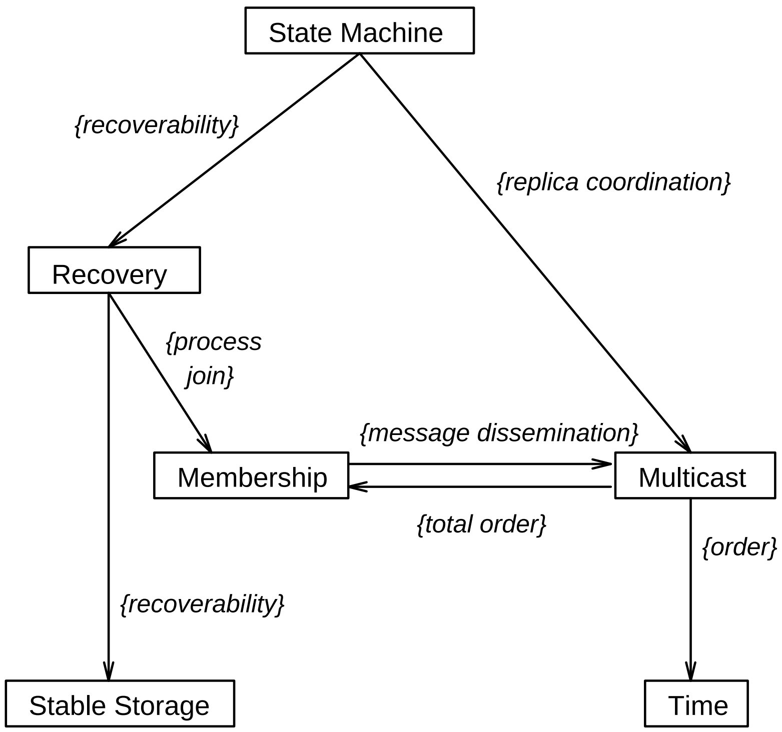

Figure 1: Fault-tolerant services and dependencies

Messages may be lost or delivered out-of-order, but it is assumed that they are never corrupted. Furthermore, machines are assumed to suffer *fail-silent* semantics [PSB$^+$88], i.e., they fail by crashing without making any incorrect state transitions. Finally, Consul assumes that stable storage is available to each machine, and that data written to stable storage survives crashes [Lam81].

In keeping with our emphasis on modularity, the mapping from fault-tolerant services to protocols is primarily 1-to-1 or 1-to-few; that is, the services are implemented independently of one another as individual protocols or a small set of protocols, rather than together in one monolithic system. Figure 2 illustrates the detailed architecture of a typical Consul protocol configuration. In this figure, the rectangles are protocols, with an arrow from protocol P_1 to protocol P_2 indicating that P_1 invokes operations on P_2 to implement its functionality. The mapping from fault-tolerant service to protocols is as follows. The Recovery protocol implements the recovery service, the Membership and FailureDetection protocols collectively implement the membership service, and a combination of the Psync and Order protocols implement the multicast and time services. In this figure, the stable storage and network protocols are shaded to indicate that they are provided externally, and hence, assumed by Consul.

Psync is the main communication mechanism in Consul [PBS89]. It provides a multicast facility that maintains the partial order of messages exchanged in the system. Specifically, it supports a *conversation* abstraction through which a collection of processes such as the state machine replicas exchange messages. A conversation is explicitly opened by specifying a set of participating processes called the *membership list*, ML. A message sent to the conversation is multicast to all processes in ML. Fundamentally, each process sends a message in the *context* of those messages it has already sent or received, a relation that defines a partial ordering on the messages exchanged through the

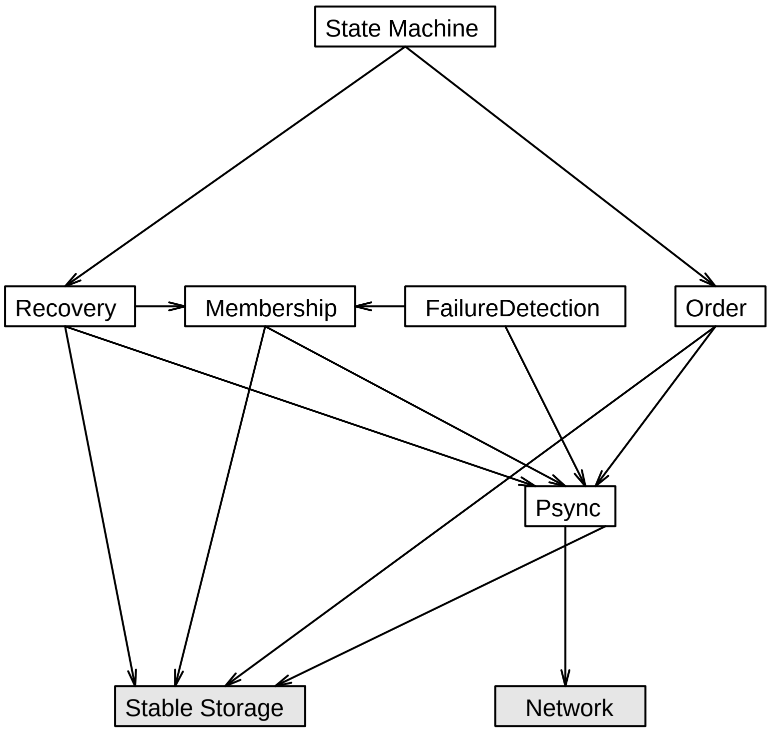

Figure 2: Consul protocol configuration

conversation. Psync explicitly maintains the partial order, which has also been called *causal order* [BSS91], in the form of a directed acyclic graph called a *context graph*. Psync provides operations for sending and receiving messages, as well as for inspecting the context graph. The multicast message delivery implemented by Psync is atomic, i.e., either all the processes in ML receive the message, or none do.

Consul's Order protocol enforces consistency on the order in which replicas receive messages, a property that is used to guarantee that replicas process commands in consistent way. This protocol is chosen from a suite of different and independent protocols, each providing a different kind of consistent message ordering using the partial ordering provided by Psync as a base. To date, we have designed two Order protocols. One is a consistent total order; when combined with the atomic message delivery guarantees of Psync, this gives the effect of an *atomic broadcast* [CASD85, KTHB89, MSM89, VRB89]. The other is a semantic dependent order; this takes advantage of the commutativity of the commands encoded in messages to provide an ordering that is less restrictive than total ordering, yet still strong enough to preserve the correctness of the application [MPS89].

The FailureDetection and Membership protocols deal with replica failures and recoveries [MPS91b]. The FailureDetection protocol monitors replicas for failures. It does this based on message traffic, i.e., if no message is received from some replica in a given interval of time, its failure is suspected and announced to the other replicas. The Membership protocol maintains a consistent system-wide view of which replicas are functioning and which have failed at any point in time. It does this by establishing agreement among correct replicas on (a) whether a replica that is suspected down has actually failed and (b) when that failure occurred relative to the stream of messages. Similarly, when a previously failed replica recovers, this protocol consistently incorporates it on all machines.

The Recovery protocol comes into play when a previously failed replica recovers. It restores the state of the recovering replica to the current state of the other replicas, and incorporates it smoothly back into the normal flow of the computation. This is done by first reading a checkpoint stored by the replica during execution, and then using an automatic replay of messages stored in Psync's context graph to process missing commands. Further details on all of these protocols can be found in [Mis91].

One of the fundamental problems in designing a modular fault-tolerant system such as Consul is dealing with interactions and dependencies between protocols. Some of these dependencies are direct—one protocol invokes an operation on another protocol to realize its own functionality. The edges in Figure 2 represent this type of interaction in Consul. For example, one of the tasks of the Recovery protocol is to incorporate a recovering process in the membership list before it starts operating. Recovery interacts directly with Membership to do this. Another example occurs between the Order and Psync protocols. In this case, Order invokes a Psync operation to determine if a message has been received by all functioning processes.

Unfortunately, such interactions do not capture all of the ways that protocols affect each other. Often, a protocol implicitly relies on another protocol to establish a certain property or to reach a given state within a certain amount of time. For example, by asking Psync whether a message has been received by all replicas, the total Order protocol is indirectly relying on the Membership protocol, which invokes a Psync operation that changes the membership list. Another example involves the timeout value used by the FailureDetection protocol to limit the time in which a process must send a message so as not to be suspected of having failed. While this timeout is confined to this one protocol, whether or not it is set to an appropriate value changes based on the state of other protocols. In this case, if the Membership protocol is actively processing the return of a previously failed process, the timeout interval for that process should be lengthened since communication may be delayed while it restores its state. The effect here is again indirect, this time on the performance of the system rather than its correctness. We consider the ramifications of both types of dependencies in a later section.

IMPLEMENTATION MODULARITY

Whereas the previous section outlined the abstract services offered by Consul and described the protocol suite that provides these services, this section presents a case study of how Consul's protocols were actually implemented in a particular object-oriented infrastructure—the *x*-kernel. The primary goal of the section is to illustrate how the design modularity is preserved in the actual implementation, and to highlight some of the aspects of the implementation that required special care due to the emphasis on retaining modularity. The latter include the way in which the specific collection of protocols required for an application is configured, how the substrate is initialized upon application startup, message flow, and restoration of the substrate following failure.

For the record, the Consul implementation consists of approximately 10,000 lines of C code, of which 3,500 is Psync. As already mentioned, the implementation vehicle is the *x*-kernel, an operating system kernel designed explicitly for experimenting with communication protocols. Consul currently uses a version of the *x*-kernel that runs standalone on Sun-3 workstations; a port to a version running on the Mach microkernel is in progress. Two small prototype applications have been constructed, a replicated directory server and a replicated word search game; following the completion of the Mach port, Consul will also be used to implement a replicated *tuple space* for a fault-tolerant version of the Linda coordination language [ACG86].

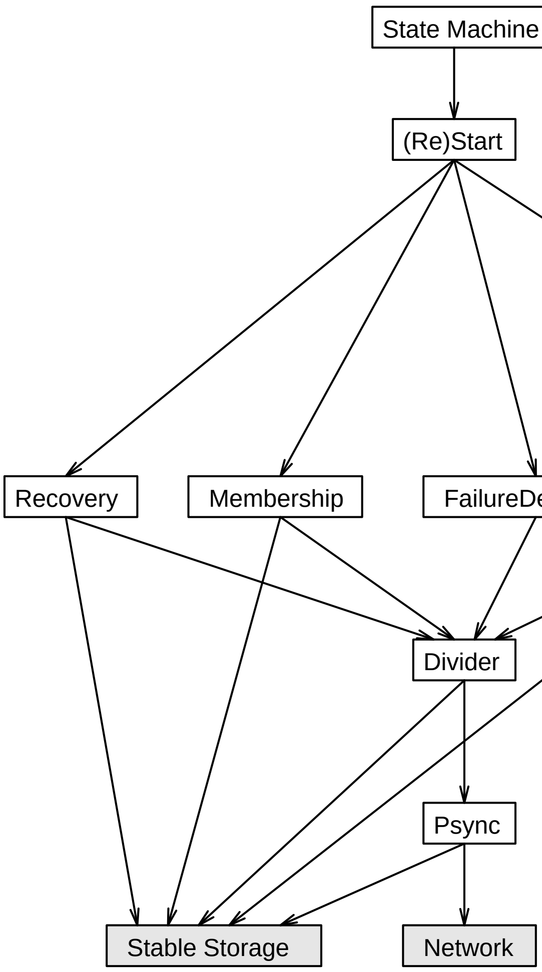

Figure 3: Consul protocol configuration

Configuration

The *x*-kernel provides an object-oriented framework designed to support the rapid implementation of efficient network protocols. It does this by providing a uniform protocol interface and support library that allows the programmer to configure individual *protocol objects* into a *protocol graph* that realizes the required functionality. Each node in this graph corresponds to a protocol object, and the edges represent a "uses" relationship. That is, an edge from protocol P_1 to protocol P_2 indicates that P_1 opens P_2 to send and receive messages on its behalf. Note, however, that in the case of Consul, the actual message flow has been optimized and hence, does not always follow these edges. This is discussed in more detail below.

Figure 3 illustrates the protocol graph that implements Consul, where each protocol described in the preceding section is implemented as an *x*-kernel protocol object. Accordingly, there are single protocol objects implementing Psync, Membership, FailureDetection, and Recovery; there is a set of objects implementing each Order protocol. In addition, the Dispatcher protocol maps messages onto commands—it tags each outgoing message with a command tag and calls the application-level command corresponding to the tag on each incoming message. Two final protocols—Divider and (Re)Start—are required for configuration purposes. Divider demultiplexes incoming messages to the appropriate high-level protocols; its specific role is described in more detail below. (Re)Start establishes a connection among various protocols needed by an application for proper communication, and reestablishes them after a failure; this protocol remains quiescent at other times. Neither protocol object is a real protocol in the sense that it exchanges messages with a peer on another machine; they only exist to "manage" the protocol graph. As before, the stable storage and network protocols represent facilities that are provided externally.

Opening Connections

The modular construction of Consul means that connections among the various protocols, as well as between the application and Consul, must be explicitly created at initialization time. From the perspective of the application, this occurs when the state machine replicas, each identified by a well-known port, decide to open connections among themselves in order to exchange messages. To do this, each replica process opens the top-most object in its protocol graph, specifying the well-known port and host addresses of the other replicas. This protocol then opens lower-level protocols, and so on. When a high-level protocol object opens a low-level protocol object, the low-level protocol returns an *x*-kernel *session object*. This session object represents the end-point of a connection and can be used by the high-level protocol object to send and receive messages.

Consider now the process of opening connections through the protocol graph in more detail. First, the replica process opens the (Re)Start protocol object multiple times—once for each command supported by the replica. The (Re)Start protocol then opens the Divider protocol which, in turn, opens the Psync protocol. Because there is a one-to-one relationship between the replica and the corresponding Psync session, the session identifier returned by Psync serves as an internal replica id; it is passed as an argument when each subsequent protocol is opened to ensure that the appropriate sessions are properly connected. (Re)Start then opens the Recovery, FailureDetection, and Membership protocols exactly once, and the Dispatch protocol once for every application open invocation (i.e., once for each replica-level command) and the Dispatch protocol in turn opens Order exactly once. Finally, the Recovery, Membership, FailureDetection, and Order protocols each open the Divider protocol. Divider knows which Psync session to associate these protocols with because of the unique id mentioned above.

Sending and Receiving Messages

Now consider how messages flow through the session objects that represent the dynamic configuration of Consul. Outbound messages generated by the replica process—which are the commands issued by the replica—are first sent to the Dispatch session that corresponds to the command being issued. The Dispatch session tags the message with a command id, and sends it out through the Order session. Finally, Order sends the message to the Psync session, which multicasts it over the network. Notice that the Divider never processes outgoing messages. For inbound messages that correspond to incoming commands issued by other replicas, the Psync session hands the message to the Divider protocol, which passes it up to Order. Order then delivers it to Dispatch, which finally invokes the appropriate command in the replica process. Notice that in both cases, the (Re)Start protocol does not process messages; it only exists to manage the process of opening protocols.

The flow of messages just described corresponds to commands that are issued and received by the replica process. In addition, the other protocols in Consul exchange messages with their peers on other machines. For example, the Membership protocol exchanges messages to reach agreement that a replica has failed. Also, FailureDetection receives a copy of all incoming messages. This is how it learns that remote replicas are still functioning. In short, each incoming message is potentially received by multiple protocols.

The fact that a set of protocols on the same machine have to cooperate so closely means that they need to have common knowledge about what messages look like, i.e., they share message representation information. Specifically, all protocols except for Psync recognize two types of messages: OT (operation type) message and MT (monitoring type) messages. Each OT message is a replica-generated message that invokes a specific command on each replica. MT messages, on the other hand, are used internally by Consul's protocols to exchange information. In other words, OT and MT messages roughly correspond to data and control messages in a traditional monolithic protocol, the only difference being that they are shared by a set of protocols.

Each protocol object receives one or both message types. For example, the Order protocol object receives both the OT and MT messages, while the Membership protocol object receives only MT messages. The protocol objects specify which messages they expect to receive to the Divider protocol object upon initialization, and the Divider protocol object, in turn, delivers the appropriate messages as they are received. As already noted, this means that the Divider may deliver a single incoming message to multiple high-level protocols.

Restoring Connections

Another consequence of Consul's modular construction is that machine failures cause the connections among various protocol and session objects to be lost in addition to their states. As a result, when a replica recovers, all of these objects and interconnections must be recreated. To restore these connections, every protocol and session object stores information in the stable store at a well-known logical address. Typically, a protocol object stores the number of its associated session objects, and for every one of its sessions, the logical addresses in the stable store where that session's state is checkpointed, while each session object stores its state. This is performed during the periodic checkpointing that every session performs while the system is operating. After this checkpoint is read, connections among protocol and session objects are restored by the (Re)Start protocol.

There is, however, an additional complexity that must be dealt with: the session states cannot be fully restored given only the information stored by the previous incarnation of the session object since these states also depend on the checkpoints taken by the other protocols. This problem is solved as follows. First, the (Re)Start protocol gathers the relevant checkpoints from all the protocols; these checkpoints include the internal replica id. The (Re)Start protocol then instructs the Divider protocol to restore the sessions corresponding to each unique id. The Divider protocol, in turn, invokes the Psync protocol object to reconstruct the session state corresponding to the session identifier retrieved from stable storage. The Psync protocol object creates a Psync session, reconstructs the context graph from the stable storage and returns the new unique id to the Divider protocol, which returns it to the (Re)Start protocol. (Re)Start then invokes the FailureDetection, Membership, Dispatch, and Recovery protocol objects to recover their appropriate session states, while the Dispatch protocol in turn invokes the Order protocol to recover the state of each of its sessions.

This completes restoration of the connections among various protocol and session objects of the communication substrate. The connection between the application process and the substrate is restored when the application invokes the (Re)Start protocol with the appropriate port.

DISCUSSION

In this section, we turn from describing how modularity has been used in the design and implementation of Consul, to discussing three specific issues that arose with respect to modularity in the course of our efforts. The first is simply defining the modules and their interfaces. This can be viewed as two steps: (a) identifying dependencies between fundamental functions and deciding whether the interactions are clean enough to allow implementation as separate modules, and (b) deciding which interactions can (or should) be implemented explicitly as part of the interface. The second is dealing with problems that arise when protocols that are correct and efficient in isolation are configured into a system that turns out to be neither. This can be viewed as dealing with the indirect interactions—mostly unexpected—that arise when dealing with the asynchrony and uncertainty of networks and failures. The third concerns our experiences with the underlying object infrastructure.

Defining Modules and Interfaces

A crucial first step in building a modular system is determining how to divide the required functionality into separate modules, and then defining the appropriate interfaces. Ideally, the goal of this process is to isolate each fundamental function in a module, where a fundamental function may be defined informally as one that is needed by multiple other modules. Unfortunately, identifying such functions can be non-trivial in a fault-tolerant distributed system, mainly because it requires identifying and understanding the nature of the direct *and* indirect interactions between modules. Until recently, even the interactions among the abstract fault-tolerant services have been poorly understood, to say nothing of finer grain pieces.

To illustrate the difficulty in doing this, consider the membership service in Consul. As already noted, we have identified two individual functions that can be combined to realize membership: detecting process failures and establishing agreement on the order of failures and joins of various processes. The first is implemented in Consul by the FailureDetection protocol and the second by the Membership protocol. One dependency here is that the FailureDetection protocol needs to initiate the Membership protocol upon suspecting the failure. This one is relatively easy to implement as a direct interaction: FailureDetection invokes an "I suspect replica R has failed" operation on the Membership protocol.

A more subtle dependency, however, comes from the fact that the agreement algorithm in the Membership protocol is affected by subsequent process failures during its execution. As a result, these functions have been coalesced into a single protocol in many systems [CM84, BJ87]. For some applications, however, the functionality provided by the FailureDetection protocol alone is sufficient to maintain correctness in the presence of failures. Separating the two functions, as was done in Consul, therefore allows the user to choose between two types of membership services: a weak membership service based only on the FailureDetection protocol, and a strong membership service based on failure detection and agreement. Moreover, our experience is that this division also makes it easier to implement the individual pieces since they become smaller and more self-contained.

Psync provides another example—in this case, a negative one—of how dependencies between functions complicate the process of defining modules. This protocol involves several different functions: it maintains the context graph, supports a consistent causal ordering, implements atomic delivery of messages, and provides various graph inspection operations. Moreover, there are many applications that would benefit from separating these functions into individual protocols since they require only a subset of the complete set; for example, certain read-only databases require atomic delivery but not consistent causal ordering [GMS91]. Unfortunately, at the time Psync was designed and implemented, we were only beginning to explore the nature of the dependencies between these functions and the benefits of modularity, so it ended up as a single, rather monolithic, protocol. Our understanding has progressed to the point where we are currently considering reimplementing Psync in a more modular fashion.

Going hand-in-hand with dividing system functionality into modules is the problem of actually defining module interfaces, that is, determining what the direct interactions should be and how they should be realized. The primary problem is that the nature of these direct interactions evolve and change as the system is constructed and as the modules are used in new ways. We experienced this many times during the design and implementation of Consul. For example, Psync was originally designed for an environment in which machines did not fail, meaning that there was no support for managing process failures or recoveries. When we started to consider such scenarios, it became necessary to return to the design and add new primitives to Psync's interface to, for example, allow the Membership protocol to modify the membership list, which is actually maintained by Psync.

Another aspect of the Psync interface that we are currently reevaluating is how a high-level protocol receives a message from Psync. We have considered two possibilities: Psync provides an operation that can be invoked to receive a message, and the high-level protocol provides a callback procedure that Psync invokes when a new message arrives. The former has the advantage that the application has the control of when a message is received and processed. In particular, since the high-level protocol decides when to receive a message, it does not need to worry about synchronization problems that occur when multiple messages arrive at the same time. The latter possibility, however, is more efficient because a message (thread) hand-off need not occur. Currently, the interface includes both styles and the high-level protocol chooses the desired one at initialization time.

A final interface example deals with Membership. Currently, this protocol removes failed processes from the membership list and reincorporates recovered processes, but it does not generate a failure notification event to inform the application about the membership change. This type of membership has been called a *monitor service* elsewhere [VM90]. However, like other researchers investigating membership protocols [BJ87, CM84, Cri88, KGR91], we now recognize that certain applications may desire such a notification to make certain application-level decisions. We are in the process of modifying the interface of the Membership protocol to provide this type of user-level notification.

Combining Modules

As noted in the Introduction, one of the advantages of modularization is that it makes it easier to develop and test the individual system functions in isolation. Unfortunately, our experience has been that even protocols that are correct and efficient by themselves sometimes become incorrect or inefficient when combined with other protocols. Unforeseen indirect dependencies are most often the cause for these problems, many of which can be traced to either the asynchronous nature of the network or the effects of failures. Said another way, having to accommodate asynchrony and failure is what makes fault-tolerant, distributed systems difficult to modularize.

One example of this occurred in Consul as a result of checkpointing requirements. In order to correctly recover from failure, every protocol must periodically write a checkpoint to stable storage. This requirement is, of course, easy to realize when working on a module in isolation, but a problem arises when multiple modules are configured into a system. In particular, some of the checkpoints need to be coordinated, leading to subtle indirect dependencies between protocols. For example, the Membership protocol checkpoints the set of live processes, while the Order protocol checkpoints the messages that have been processed by the application. Since any change in the membership list depends of the set of messages that have been processed—in this case Membership messages—some coordination is needed between these two protocols to do checkpointing correctly, causing a dependency between them. Another checkpointing dependency is between the state machine replica and the Order protocol. The checkpoint taken by the replica represents the state derived from the set of messages it has processed, while the checkpoint taken by the Order protocol contains the messages themselves. Clearly these two checkpoints must be coordinated to preserve correct semantics.

The way checkpoint coordination is actually realized in Consul differs from what might be expected. Ideally, the modules involved in the coordinated action would communicate among themselves before every checkpoint. However, in the absence of any direct support for such communication in the underlying object infrastructure, the protocols in Consul actually perform their checkpointing based on the state of the Psync context graph. For example, the Membership protocol and the Order protocol take their checkpoint every time it is known that the order of messages in a portion of the context graph—specifically, a *wave*—cannot be altered by future message arrivals. Since any change in the membership is reflected when the wave containing the membership initiation message is in this state, this scheme is sufficient. However, there is a price to be paid here in that the criteria for checkpointing depends on the functionality of the modules, and the frequency of the checkpointing has to be fixed in advance and cannot be varied arbitrarily.

A similar class of indirect dependencies that were only discovered when modules were combined relates to timer management. The specific problem here is that the optimality of the various timer values used by protocols is affected by the execution of others. One example concerns the FailureDetection protocol, which uses a timer to put a limit on the length of time that can pass before a process on another machine is suspected to have failed. The value of this timer has great effect on the performance of the entire system. A large value implies that a process failure may not be detected until long after it occurs, potentially causing lengthy delays in the the application; a small value, on the other hand, increases the possibility of false failure suspicions. Unfortunately, choosing an optimal value depends on factors outside of the FailureDetection protocol itself, such as whether the Membership protocol is being executed, whether recovery of some process is in progress, whether Psync is doing a message retransmission, or how fast the application is executing. Hence, the protocols managing these factors indirectly affect the correctness and performance of the FailureDetection protocol. A similar situation occurs with the timer used by Psync to signal when messages should be logged onto stable storage. A smaller value of this timer increases the logging overheads while a larger timer value implies that a large number of messages may have to be retransmitted during recovery, thus increasing the recovery cost. The optimal value of this timer depends, in general, on the rate at which messages are exchanged in the system, which in turn depends on how fast the replicas are executing.

Active coordination between modules to set timer values in Consul is limited, so optimal delay intervals are not always realized. Nevertheless, an attempt is made to keep values close to optimal by modifying them according to what modules learn about the system state by monitoring the messages exchanged. For example, the FailureDetection protocol receives all messages in the course of implementing its failure detection duties, thereby allowing it to infer when a process is recovering and increase the appropriate timer value. Note that this strategy does not work in all cases. For example, another situation where an increase in timer value is warranted occurs when Psync is in the midst of retransmitting a message. This cannot be detected by the FailureDetection protocol, however, and so it cannot adjust its timer accordingly. The net result is an increased danger of false failure detections until normal processing is resumed.

Underlying Object Support

The underlying object infrastructure used to implement a modular fault-tolerant distributed system such as Consul has great effects on its feasibility and performance. This section identifies the successes and failures we experienced in implementing the Consul protocols in the *x*-kernel. In particular, since the *x*-kernel was originally designed to support traditional network protocols—generally one-to-one protocols that do not attempt to recover from processor failures—the interesting question is how well the *x*-kernel served as a platform for implementing fault-tolerant group/multicast protocols.

Common Support Routines. One of the main contributions of the *x*-kernel is a set of library support routines that protocol and session objects call to do the work necessary to implement their functionality. That is, the *x*-kernel recognizes a set of operations common to all protocols, and provides them as library routines rather than forcing protocol implementors to implement their own. We recognized three important functions common to a majority of the Consul protocol suite.

First, Consul protocols use messages to communicate with each other. For example, the FailureDetection protocol, upon suspecting a process failure, sends a message to the Membership protocol. These two protocols share knowledge about the structure of such a message, and as a result, they can decode each other's messages. In general, typical message operations include creating and destroying a message, reading and copying a message, breaking and reassembling messages, and appending and deleting message headers. These operations tend to be quite complex and hinder the modular development. Hence, system support in the form of message library providing these operations greatly simplifies module implementation. Because the *x*-kernel was designed to support protocol implementations, and all protocols manipulate messages, the *x*-kernel provides adequate message support for Consul.

Second, timers are fundamental to fault-tolerant computing systems. In Consul, timers are used by Psync, FailureDetection, and the Order protocols—Psync uses a timer to manage the logging of messages onto the stable storage, FailureDetection uses timers to monitor other processes, and the various Order protocols use timers to checkpoint messages delivered to the application. As with the message library, the *x*-kernel provides sophisticated timer support that proved useful in implementing Consul.

Third, fault-tolerant protocols depend on a stable storage facility to recover after a processor failure. In Consul, for example, Psync stores the context graph and the Order protocols store the set of messages that have been delivered to the application in stable storage. In addition, every protocol checkpoints the state of its active sessions in the stable storage. Upon recovery, this information is retrieved by these protocols to reconstruct their sessions. The *x*-kernel, because it was designed to support network protocols that typically do not attempt to maintain connections across host failures, does not provide a stable storage facility. As a result, we had to construct such a facility from scratch. Extending the *x*-kernel to provide a stable storage facility analogous to its message and timer facilities would make the *x*-kernel better suited for implementing fault-tolerant protocols.

Object Interaction. Clearly, it is important for an object infrastructure to provide a mechanism for instantiating objects and for establishing links between objects so that they can invoke operations on each other. The *x*-kernel, because it supports both static protocol objects that are defined at configuration time, and dynamic session objects that are created at connection establishment time, provides a reasonable base for implementing Consul. On the other hand, the *x*-kernel provides a somewhat limited model for protocols to interact (communicate) with each other, and this limitation greatly affected how we implemented Consul. Consider the following two problems, and how we worked around them.

First, the *x*-kernel defines a simple interface between protocols that includes operations for opening and closing connections, and sending and receiving messages. Any additional operations that one protocol might want to export to another are encapsulated in a control operation, not unlike the Unix ioctl system call. Traditional network protocols use control operations to ask each other a limited number of well-defined questions, for example, about optimal packet sizes and peer addresses. The Consul protocols, however, require a much richer interface. For example, high-level protocols can invoke a large collection of operations on Psync to query the state of the context graph. All of these operations had to be folded into the control operation. An ideal object infrastructure would give all such operations first class status.

Second, the *x*-kernel was designed to support a mostly linear composition of protocols, i.e., a protocol stack where one high-level protocol depends on one low-level protocol. In Consul, however, there exist several protocols "at the same level" that are all cooperating to implement a set of services. For example, Membership, Order, and FailureDetection all cooperate with each other, but without being stacked one on top of the other. This expectation that protocols are composed in a linear fashion is, in fact, the most fundamental limitation of the *x*-kernel. We were able to work around this problem in Consul because all the cooperating protocols shared a single Psync session, and since Psync is a multicast protocol, it reflected all outgoing messages back up the protocol graph, in addition to out over the network. In this way, any message sent by FailureDetection is seen by the local Membership protocol, in addition to all the copies of the Membership protocol running on remote machines. Similarly, as mentioned above, multiple protocols coordinate their checkpoints based on the state of the Psync context graph they all share. In other words, the Consul protocols needed to be linked together with a "broadcast channel" so they can all see each other's messages; Psync, rather than the *x*-kernel, had to provide this functionality in Consul.

CONCLUSIONS

In attempting to "push the envelope" with respect to modularity in the design and implementation of Consul, we have learned a number of lessons. Perhaps first and foremost is that, despite the difficulties described in the previous section, our experience reinforced the idea that modularity is a powerful tool for simplifying the construction of complex software systems. The design, implementation, and testing were all easier given that each function was isolated from the rest of the system rather than being a small piece of a larger whole. The ability to configure the functions into a customized platform for a particular application is also a significant advantage.

The second general lesson concerns dependencies between modules. Specifically, we believe that understanding the precise nature of the dependencies between modules and accounting for their effects is crucial to producing a correct and efficient system. Dependencies resulting from direct interactions are in general easier to deal with, implying that every effort should be made to program interactions explicitly whenever possible. Unfortunately, one of the fundamental difficulties in modularizing systems of this type is that it is often impossible or impractical to do away with indirect dependencies completely due to the asynchronous nature of distributed systems and the effect of failures. Indeed, even anticipating all of the situations that can lead to indirect dependencies is a significant problem.

Finally, our experience provided valuable feedback on the properties required of an object infrastructure designed to support modularity in the implementation. For example, the infrastructure needs to provide for a rich "interconnect" over which the modules share information. Based on this experience, work is underway to develop a new framework for fault-tolerant protocols that better facilitates modularization [HS92].

ACKNOWLEDGMENTS

This work has been supported in part by NSF Grant CCR-9003161, ONR Grant N00014-91J-1015, and DARPA Contract DABT63-91-0030.

REFERENCES

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[BJ87] K. Birman and T. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, Feb 1987.

[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.

[CDD90] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in the Advanced Automation System. In *Proceedings of the Twentieth Symposium on Fault-Tolerant Computing*, pages 6–17, Newcastle-upon-Tyne, UK, Jun 1990.

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

[Cri88] F. Cristian. Agreeing on who is present and who is absent in a synchronous distributed system. In *Proceedings of the Eighteenth International Conference on Fault-tolerant Computing*, pages 206–211, Tokyo, Jun 1988.

[Cri91] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of ACM*, 34(2):56–78, Feb 1991.

[GMS91] H. Garcia-Molina and A. Spauster. Ordered and reliable multicast communication. *ACM Transactions on Computer Systems*, 9(3):242–271, Aug 1991.

[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.

[HS92] M. Hiltunen and R. Schlichting. Modularizing fault-tolerant protocols. In *Fifth SIGOPS European Workshop*, Le Mont Saint-Michel, France, Sept 1992. To appear.

[KDK+89] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger. Distributed fault-tolerant real-time systems: The Mars approach. *IEEE Micro*, pages 25–40, Feb 1989.

[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.

[KM85] H. Kopetz and W. Merker. The architecture of MARS. In *Proceedings of the Fifteenth Symposium on Fault-Tolerant Computing*, pages 274–279, Ann Arbor, Mi, Jun 1985.

[KTHB89] M. F. Kaashoek, A. Tanenbaum, S. F. Hummel, and H. Bal. An efficient reliable broadcast protocol. *Operating Systems Review*, 23(4):5–19, Oct 1989.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of ACM*, 21(7):558–565, Jul 1978.

[Lam81] B. Lampson. Atomic transactions. In *Distributed Systems—Architecture and Implementation*, pages 246–265. Springer-Verlag, Berlin, 1981.

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

[MPS89] S. Mishra, L. Peterson, and R. Schlichting. Implementing replicated objects using Psync. In *Proceedings of the Eighth Symposium on Reliable Distributed Computing*, pages 42–52, Seattle, Washington, Oct 1989.

[MPS91a] S. Mishra, L. Peterson, and R. Schlichting. Consul: A communication substrate for fault-tolerant distributed programs. Technical Report TR 91-32, Dept of Computer Science, University of Arizona, Tucson, AZ, 1991.

[MPS91b] S. Mishra, L. Peterson, and R. Schlichting. A membership protocol based on partial order. In *Proceedings of the Second Working Conference on Dependable Computing for Critical Applications*, pages 137–145, Tucson, AZ, Feb 1991.

[MS92] S. Mishra and R. Schlichting. Abstractions for constructing dependable distributed systems. Technical Report 92-19, Dept of Computer Science, University of Arizona, Tucson, AZ, 1992.

[MSM89] P. M. Melliar-Smith and L. E. Moser. Fault-tolerant distributed systems based on broadcast communication. In *Proceedings of the Ninth International Conference on Distributed Computing Systems*, pages 129–134, Newport Beach, CA, Jun 1989.

[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.

[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.

[VRB89] P. Verissimo, L. Rodrigues, and M. Baptista. Amp: A highly parallel atomic multicast protocol. In *SIGCOMM'89*, pages 83–93, Austin, TX, Sep 1989.