Modularity in the Design and Implementation of Consul

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

TR 92-20

Abstract

Many applications constructed as Autonomous Decentralized Systems require high dependability, often leading to the use of distributed architectures and their associated fault-tolerance techniques. Consul is a system designed to support the use of such techniques in the construction of fault-tolerant, distributed systems structured according to the state machine approach. Here, the way in which modularity has been used in the design and implementation of Consul is described. Our approach to this issue makes it easy to configure a system customized to the needs of a specific application, as well as facilitating the development of the individual components that make up Consul.

August 3, 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

1 Introduction

Distributed computer systems, in which multiple machines are connected by a communication network with no shared memory, are important for realizing many of the potential benefits of Autonomous Decentralized Systems (ADS). For example, the lack of memory contention in such an architecture makes distributed systems more scalable than their shared memory counterparts, thus facilitating system expandability. Another advantage, which is the focus of this paper, is that distributed systems provide an inherent redundancy that can be exploited to improve the overall dependability of the system [Lap92]. High dependability is often the single-most important attribute needed in many of the applications for which ADS are used, such as railway control and manufacturing systems.

To achieve high dependability, the software for a distributed system must be implemented as a *fault-tolerant, distributed program* that can continue executing despite processor or network failures. Recently, we have designed and implemented a system called Consul [Mis91, MPS91b] that provides support for constructing such programs using the *state machine approach* [Sch90]. It does this by providing 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 in this approach. These services are realized using *protocols* as the fundamental modules of the system. A large amount of research has been performed in areas related to this approach, including development of new algorithms [CM84, Cri88, GMS91, RB91, VM90] and systems [BSS91, KDK$^+$89, PSB$^+$88].

In this paper, we overview the way in which modularity has been used in the design and implementation of Consul. Unlike other 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. This configuration process is relatively simple, which makes it feasible to customize the system by selecting only those specific services required by the application. In addition, constructing a system from fundamental building blocks in this way also makes it easier to design, implement, debug, and optimize individual protocols, as well as facilitating the identification of interactions and dependencies among protocols. Of course, it has long been recognized that judicious use of modularization can simplify the development of any type of complex system. Our goal during the development of Consul has been to bring this technique to bear on the problem of constructing fault-tolerant, distributed systems.

This paper is organized as follows. In Section 2, we first outline the basics of the state machine approach and then describe the design of Consul, focusing on the various individual protocols that have been defined. Section 3 then describes the structural aspects of the system implementation; in addition to serving as a case study of how such modularity is achieved in fault-tolerant systems, this description also documents other interesting features of the implementation. Our use of the *x*-kernel [HP91] as an implementation platform has been influential in many aspects of Consul's implementation, so its fundamental characteristics are outlined here as well. Finally, Section 4 offers some conclusions.

2 Design Modularity

As already noted, Consul is a collection of protocols that form a communication substrate upon which fault-tolerant, distributed systems can be built using the state machine approach [Sch90]. In this approach, a system is structured as a collection of generic services that are implemented by multiple processes for fault-tolerance. Each such service is characterized as a state machine, which maintains *state variables* that are modified in response to *commands* that are received from other state machines or the environment. Execution of a command is deterministic and atomic with respect to other commands.

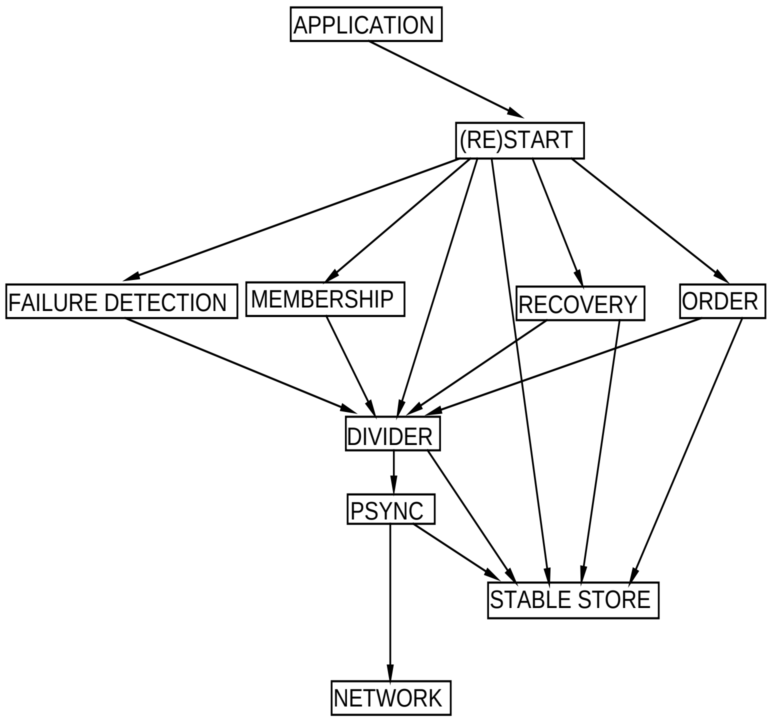

Figure 1: Consul protocol configuration

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 that 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 protocols 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.

Consul assumes a distributed architecture in which the communication network is asynchronous, *i.e.*, there is no bound on the transmission delay for a message between any two machines. Messages may be lost or delivered out-of-order, but it is assumed that they are never corrupted. Furthermore, processors 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 processor, and that data written to stable storage survives processor crashes [Lam81].

A copy of Consul resides on each processor in the system, and provides an interface between the application program (*i.e.*, the state machine replicas) and the underlying network. The system architecture is optimized to handle a relatively small number of replicas, a reasonable strategy given Consul's orientation towards the state machine approach. It does mean, however, that issues of scale have not been addressed explicitly as they have been in other systems [BSS91, GMS91].

In keeping with our emphasis on modularity, the fault-tolerant services supported by Consul are implemented independently of one another as individual protocols rather than together in one monolithic system. Figure 1 illustrates the detailed architecture of a typical protocol configuration in Consul. In this figure, the rectangles are protocols, with an arrow from protocol P_1 to protocol P_2 indicating that P_1 depends on the correct functioning of P_2 to ensure its own correctness [Cri91]. The stable store protocol at the base of the substrate provides a storage facility that survives processor crashes. The (re)start and divider protocols are configuration protocols, *i.e.*, they aid the user in building a system according to the requirements of the application. The (re)start protocol 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. The divider protocol is a demultiplexing protocol that directs messages in the system to the appropriate protocols. The role of these protocols is described further in Section 3.

Psync is the main communication mechanism in Consul [PBS89]. It provides a group-oriented interprocess communication mechanism in the form of 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 conversation. This partial order, which has also been called *causal order* [BSS91], is maintained explicitly by Psync in the form of a directed acyclic graph called a *context graph*. At any given time during execution, each participant has a *view* of the context graph, which is the subgraph corresponding to those messages it has sent or received up to that point. Psync provides a set of 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.

The order protocol in Consul enforces consistency among replicas on the order in which they receive messages, a property that is used to guarantee that replicas process commands in an order that maintains state consistency. This protocol is chosen from a suite of different and independent protocols, each providing a different kind of consistent message ordering using the consistent partial ordering provided directly by Psync as a base. At this point, two other kinds of orderings have been constructed. 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 failure detection and membership protocols deal with replica failures and recoveries, and together comprise Consul's membership service [MPS91a]. The failure detection protocol is used to monitor 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. 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 into ML on all machines.

The recovery protocol comes into play when a previously failed replica recovers. Specifically, it deals with restoring the state of the recovering replica to the current state of the other replicas, and incorporating 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].

3 Implementation Modularity

The implementation of Consul is made up 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. The version of the *x*-kernel currently being used by Consul executes standalone on Sun-3 workstations, with a port to a version running on the Mach microkernel in progress. Two small prototype applications have been constructed, a replicated directory object 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]. Here, we first give an operational overview of the *x*-kernel derived from [HP91] and then describe how it has been used to construct a modular implementation of Consul.

3.1 Overview of the x-kernel

The x-kernel is an operating system kernel explicitly designed to support the rapid implementation of efficient network protocols by providing a uniform protocol interface and an implementation support library. The x-kernel provides three primitive communication objects: *protocols*, *sessions*, and *messages*. Protocol objects are static and passive. Each protocol object corresponds to a conventional network protocol—*e.g.*, IP [Pos81], UDP [Pos80], TCP [USC81]—where the relationships between protocols are defined at the time a kernel is configured. Session objects are also passive, but they are dynamically created. Intuitively, a session object is an instance of a protocol object that contains a "protocol interpreter" and the data structures that represent the local state of some "network connection". Messages are active objects that move through the session and protocol objects in the kernel. The data contained in a message object correspond to one or more protocol headers and user data.

A protocol object supports three operations for creating session objects:

```
session = open(protocol, invoking_protocol, participant_set)
open_enable(protocol, invoking_protocol, participant_set)
session = open_done(protocol, invoking_protocol, participant_set)
```

Intuitively, a high-level protocol invokes a low-level protocol's `open` operation to create a session; that session is said to be in the low-level protocol's class and created on behalf of the high-level protocol. Each protocol object is given a capability for the low-level protocols upon which it depends at configuration time. The capability for the invoking protocol passed to the `open` operation serves as the newly created session's handle on that protocol. In the case of `open_enable`, the high-level protocol passes a capability for itself to a low-level protocol. At some future time, the latter protocol invokes the former protocol's `open_done` operation to inform the high-level protocol that it has created a session on its behalf. Thus, the first operation supports session creation triggered by a user process (an *active* open), while the second and third operations, taken together, support session creation triggered by a message arriving from the network (a *passive* open).

In addition to creating sessions, each protocol also "switches" messages received from the network to one of its sessions with a `demux(protocol, message)` operation. `demux` takes a message as an argument, and either passes the message to one of its sessions, or creates a new session—using the `open_done` operation—and then passes the message to it.

A session object supports two operations: `push(session, message)` and `pop(session, message)`. The first is invoked by a high-level session to pass a message down to some low-level session. The second is invoked by the `demux` operation of a protocol to pass a message up to one of its sessions.

3.2 Substrate Implementation using the x-kernel

Each of the protocols shown in Figure 1 is available to the system designer as a protocol object. Accordingly, there are single protocol objects implementing Psync, divider, membership, failure detection, and recovery; there is also a suite of objects implementing order, where each object realizes a different ordering semantics. Connections among the protocol objects are established by the (re)start protocol object. For every specific combination of protocols needed by the user, there is a separate protocol object implementing the corresponding (re)start protocol; the application picks up the appropriate (re)start protocol object that suits its needs.

When compared to a monolithic alternative, our emphasis on modularity in the implementation of Consul forced us to pay special attention to the communication and configuration aspects of the system. Accordingly, we now focus on these aspects of the implementation by describing how messages are structured, how the connections between various protocol objects are established initially and used during normal operation, and how these connections are restored following a failure. We also discuss optimizations that were performed to avoid some of the performance penalties often associated with modular systems.

Message Structure. In general, the emphasis on modularity in the design of Consul means that protocols need to interact with one another to implement a given fault-tolerant service. For example, as noted in Section 2, the failure detection and membership protocols cooperate to implement the functionality of a membership service. To facilitate this type of cooperation, both between protocols on different machines as well between different protocols on the same machine, certain information about the structure of messages is shared among protocols in the system. Specifically, information is shared about the two types of messages that are pushed onto the Psync protocol object in the communication substrate: OT (operation type) and MT (monitoring type). The OT message is an application-generated message used to invoke specific operations on each replica, while the MT message is used to ensure the consistency of Consul during failures and recoveries.

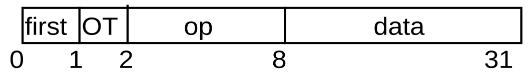

Figure 2: Operation Type message

Figure 2 schematically depicts the OT message. As shown here, it is four bytes long, with the numbers along the bottom indicating the starting position of each field. Here, *first* indicates whether this is the first message of the system, *op* is the operation to be invoked on the object and the *data* includes the arguments of the operation. The MT message, which is 12 bytes long, is schematically depicted in Figure 3. Here, *mode* indicates the type of the membership message and *p_addr* indicates the address of the replica. As described in [MPS91a], there are five types of membership messages: `<P is down>`, `<P is up>`, `<Ack, P is down>`, `<Nack, P is down>` and `<Ack, P is up>`. The first is sent by the failure detection protocol object upon suspecting the failure of a replica to initiate the agreement protocol, the second is sent by a recovering replica for the same purpose, and the final three are responses that may be generated by other functioning replicas.

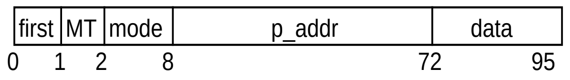

Figure 3: Monitoring Type message

In Consul, a protocol object receives one or more types of messages. 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.

Initialization. An application using Consul consists of a well-defined set of processes—one for each state machine replica—that explicitly open connections among themselves in order to exchange messages, plus potentially some user processes that submit commands externally. To establish these connections among replicas, one replica does an active open, while the remaining replicas do passive opens. This process of starting Consul is similar to that used for Psync, and in fact, an active open in Consul results in an active open of the Psync protocol object, while a passive open results is a passive open of Psync. Figure 4 shows the sessions that are created at one site in Consul; protocol objects are depicted as rectangles, with the corresponding session objects shown as circles. In the following, we describe how these sessions are created and how the connections are established among the protocol objects; in doing so, we consider both active and passive opens. For convenience, we use the terms protocol and protocol object synonymously.

Active Open. Each user process wishing to interact with the replica on a particular processor is identified by a *port_id*. This process opens the (re)start protocol once for every operation exported by the replica. This is done using the `open` primitive provided by the *x*-kernel. The parameters of this call include the *operation_id*, *port_id*, and participant set. The session returned from this call is then used by the user process to invoke the corresponding operation by doing a `push` onto this session.

The (re)start protocol is responsible for opening every protocol needed by the user process. The `open` procedure of the (re)start protocol object opens the divider protocol when it is invoked for the first time for a given *port_id*. The arguments to this call include the *restart_id*, *port_id*, and participant set. The *session_id* of the session returned from this call acts as the unique system-wide identifier of the system and is referred to as the *system_id*; since any given instantiation of Consul contains exactly one Psync session, this identifier allows a user process to participate simultaneously in multiple applications built using the system. The (re)start protocol maintains a mapping of *port_id* to *system_id* for future reference. On receiving this *system_id*, the (re)start protocol opens the failure detection, membership, recovery, and dispatch protocol objects. The failure detection, membership, and recovery protocol

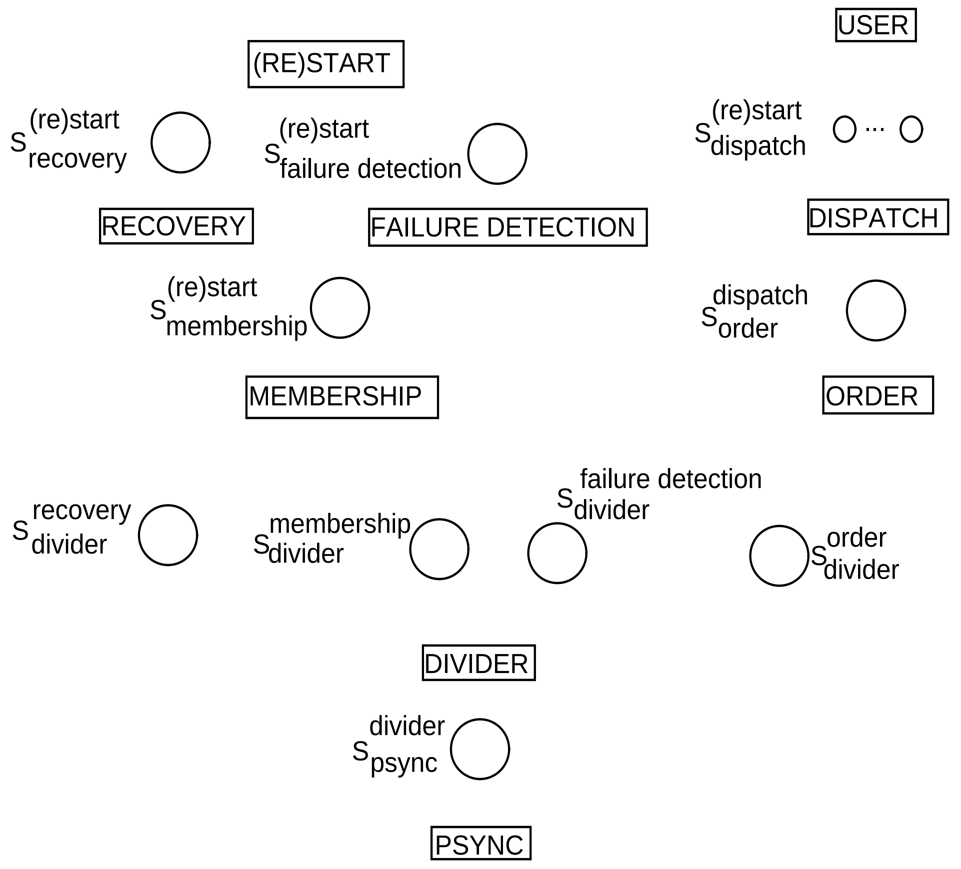

Figure 4: Protocol and session objects

objects are opened exactly once for a given *port_id*, while the dispatch protocol object is opened every time the `open` procedure of the (re)start is invoked. The arguments for all these invocations include the *system_id* and the *participant_id* of the replica invoking the call. The arguments for opening the dispatch protocol also include the corresponding *operation_id*. The session returned by the dispatch protocol is returned to the user process.

The `open` procedure of the dispatch protocol object opens the appropriate order protocol object once for a given *system_id*. It creates a session and returns it to the invoking protocol object. The pair $< S^{dispatch}_{order}, operation\_id >$ is used by the dispatch protocol to demultiplex incoming messages to the appropriate sessions above.

When the `open` procedure of the failure detection, membership, recovery, and order protocol objects is invoked, these protocol objects in turn open the divider protocol. The arguments to these calls include the *system_id* and the types of messages that these protocol objects expect to receive. These protocol objects also create a session on this invocation. The session returned by the divider protocol is used by these protocol objects to demultiplex the incoming messages to the appropriate sessions above.

The divider protocol is used to demultiplex the incoming messages to one or more protocols above. When it is opened by the (re)start protocol, it opens the Psync protocol object and returns the Psync session, returned from the Psync `open` procedure, to the (re)start protocol. When opened by some other protocol object, it creates a session and returns it to the invoking protocol. It also maintains a map from $\langle system\_id, message\_type \rangle$ to a set of sessions, which is used to demultiplex the incoming messages to the appropriate protocols. This map is updated every time a protocol other than (re)start opens the divider.

Passive Open. The user processes on the sites containing passive replicas do `openenable` on the (re)start protocol, once for every operation the replica exports. The corresponding sessions are returned by the (re)start protocol by invoking the user process' `opendone` procedure.

The `openenable` procedure of the (re)start protocol invokes `openenable` of the divider protocol exactly once for a given *port_id*. This procedure also maintains a map from *port_id* to a list of $< operation\_id, participant\_id >$ to be used when the corresponding `opendone` procedure is invoked. The `opendone` procedure of the (re)start protocol is invoked by the divider protocol when the first message is received with the *port_id* as argument. In this procedure, (re)start opens the failure detection, membership, recovery, and dispatch protocols. The `opendone` procedure then invokes the `opendone` procedure of the user process with the appropriate dispatch session as argument.

When the `openenable` procedure of the divider protocol is invoked, it performs an openenable on the Psync protocol. When the divider's `opendone` procedure is invoked, it invokes the `opendone` procedure of the (re)start protocol with the Psync session as one of the arguments. The failure detection, membership, recovery, dispatch and order protocols do not have `openenable` procedures since these protocols are always opened actively.

Optimizations. Several optimizations have been made in the implementation to improve system performance while still retaining its modular structure. The most important of these involve modifying the message flow to direct messages only to those protocol and session objects that actually process it. Figure 5 shows the path a message takes as it moves upwards from the Psync protocol through the substrate. There are two optimizations done as the message flows in this direction. First, since the (re)start protocol is needed only for establishing connections, it does not need to see messages. Accordingly the (re)start protocol object and the (re)start session object are bypassed and the message moves directly from the dispatch sessions to the user protocol object. The second optimization is bypassing the divider sessions. The divider protocol needs to see the incoming message to demultiplex it to the appropriate protocols above, but there is no function to be performed in its sessions as the message flows up. Thus, these sessions are bypassed and the message moves directly from the divider protocol to the failure detection, membership, recovery or order protocols.

Figure 6 shows the path a message takes as it moves down through the communication substrate. Again, there are two optimizations performed as the message flows in this direction. The first optimization involves bypassing the (re)start sessions. Since the (re)start sessions do not process an outgoing message, the message is directly pushed from the user to the appropriate dispatch sessions. In the second optimization, the divider sessions are bypassed. Since divider is a headerless protocol and is needed only for demultiplexing the message as it flows upwards, the corresponding sessions do not need to see the message as it moves in the other direction. Accordingly, a message is pushed directly from the failure detection, membership, recovery, or order sessions to the Psync session.

The net result of the optimizations is that the (re)start sessions are not created since they end up performing no function in the substrate. On the other hand, the divider sessions, though not used in the message flow, are created, since the *session_id* of these sessions are used by the failure detection, membership, recovery, and order protocols to demultiplex incoming messages to the sessions above.

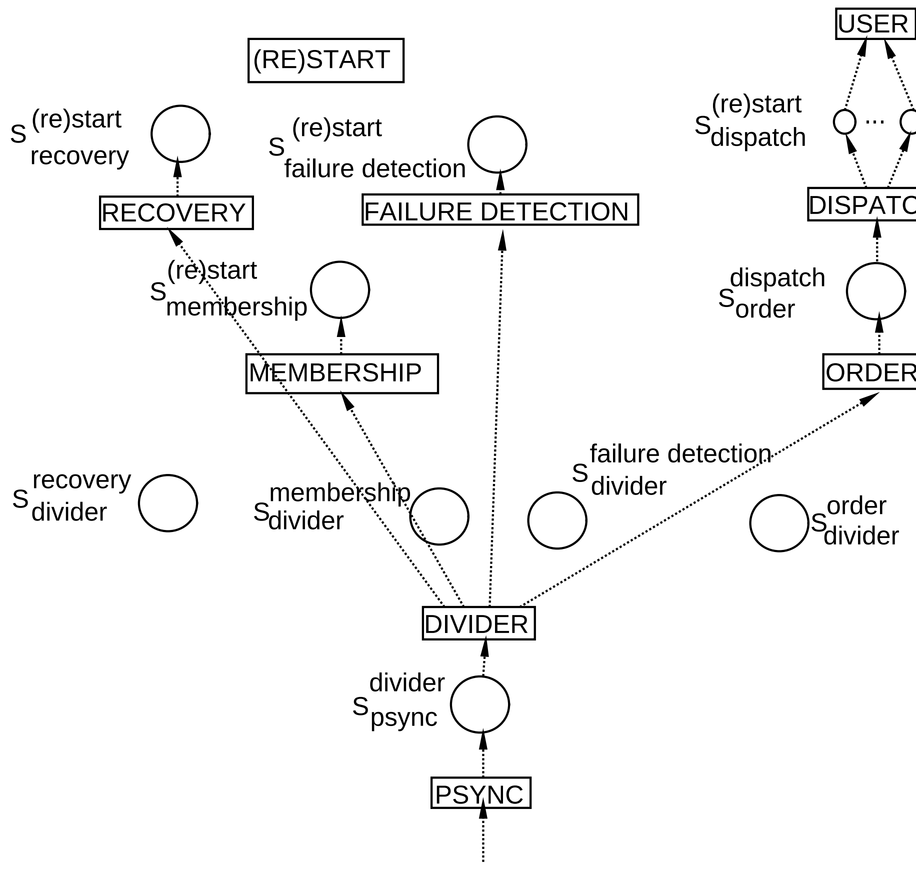
USER
(RE)START
$S^{(re)start}_{recovery}$
$S^{(re)start}_{failure\ detection}$
$S^{(re)start}_{dispatch}$
RECOVERY
FAILURE DETECTION
DISPATCH
$S^{(re)start}_{membership}$
$S^{dispatch}_{order}$
MEMBERSHIP
ORDER
$S^{recovery}_{divider}$
$S^{membership}_{divider}$
$S^{failure\ detection}_{divider}$
$S^{order}_{divider}$
DIVIDER
$S^{divider}_{psync}$
PSYNC

Figure 5: Message flow upwards

Restoring Connections. The connections among various protocol and session objects, as well as their states, are lost when a processor crash occurs. 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 session, the logical addresses in the stable store where the session state is checkpointed, while a session object stores its state. This is performed during the periodic checkpointing that every session performs while the system is in operation. After this is done, connections among protocol and session objects are restored by the (re)start protocol. However, there is an additional complexity: the session states cannot be fully restored given only the information stored by the corresponding protocol and 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 *port_id*s that invoked the (re)start protocol, the corresponding *system_id*s, and all the operation identifiers for every *port_id*. The (re)start protocol then invokes the divider protocol with a control operation to restore the sessions corresponding to each *system_id*. The divider protocol, in turn, invokes the Psync protocol object to reconstruct the session state corresponding to the *session_id* retrieved from stable storage. The Psync protocol object creates a Psync session, reconstructs the context graph from the stable storage and returns the new *system_id* to the divider protocol, which returns it to the (re)start protocol. (re)start

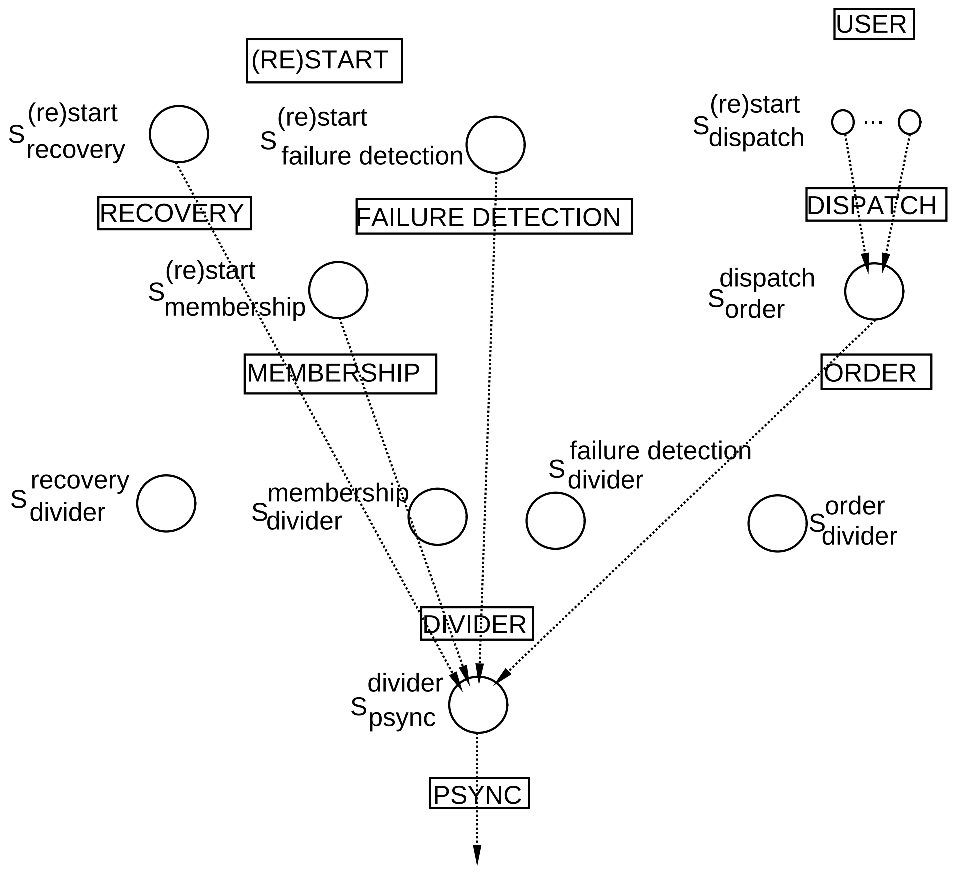

Figure 6: Message flow downwards

then invokes the failure detection, 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 its session.

This completes restoration of the connections among various protocol and session objects of the communication substrate. The connection between the user process and the substrate is restored when the user invokes the (re)start protocol with the appropriate *port_id*.

4 Conclusions

Consul is a modular system, both in its design and its implementation. The design emphasizes the use of separate protocols for each fault-tolerant service, with communication between protocols on the same or different machines being constrained to well-defined interfaces. Such a strategy avoids many of the ad-hoc structuring and control paths that often accompany a monolithic system. Moreover, the implementation retains this modularity by using the configuration and communication support provided by the *x*-kernel. Our experience, both with Consul directly [Mis91] as well as with other systems built using the *x*-kernel model [HPOA89, OP92], is that this modularization comes with little or no performance penalty.

Despite our positive experience with modularization in this context, there were, in fact, a number of difficulties that made the process less straightforward than it might appear. Many of these were caused by *indirect dependencies* between protocols. In contrast with direct functional dependencies such as those illustrated in Figure 1, an indirect dependency occurs when execution of one protocol affects another without any direct communication. For example, in Consul, the states of the recovery, membership, and Psync protocols affect the optimal value of the timer maintained by the failure detection protocol. Such dependencies complicate the process of defining modules and their interfaces, and ensuring correctness and efficiency of the system.

Our current research in this area is concentrating on identifying and characterizing the dependencies that exist between protocols used for building fault-tolerant, distributed systems. As part of this effort, we are also working on developing a new model for this type of protocol that we hope will facilitate modularization [HS92]. This new model is based on further refining protocols into their orthogonal properties and then realizing these properties with a standard system framework.

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] Sudhir Ahuja, Nicholas Carriero, and David Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[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.

[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.

[HPOA89] N. C. Hutchinson, L. L. Peterson, S. O'Malley, and M. Abbott. RPC in the *x*-kernel: Evaluating new design techniques. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, pages 91–101, Litchfield Park, AZ, Dec 1989.

[HS92] M. Hiltunen and R. D. 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.

[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.

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

[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.

[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. 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.

[MPS91b] S. Mishra, L. L. Peterson, and R. D. 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.

[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.

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

[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.

[Pos80] Jon Postel. User datagram protocol. Request For Comments 768, USC Information Sciences Institute, Marina del Ray, Calif., Aug 1980.

[Pos81] Jon Postel. Internet protocol. Request For Comments 791, USC Information Sciences Institute, Marina del Ray, Calif., Sep 1981.

[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.

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

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

[USC81] USC. Transmission control protocol. Request For Comments 793, USC Information Sciences Institute, Marina del Ray, Calif., Sep 1981.

[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.