[43] B. Walter. A robust and efficient protocol for checking the availability of remote sites. *Computer Networks*, 6(3):173–188, Jul 1982.

[32] B. W. Lampson. Designing a global name service. In *Proceedings of the 5th Symposium on Principles of Distributed Computing*, pages 1–10, Aug 1986.

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

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

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

[36] B. M. Oki. Viewstamped replication for highly-available distributed systems. Technical Report TR MIT/LCS/TR-423, MIT Laboratory for Computer Science, Cambridge, MA, Cambridge, MA, Aug 1988.

[37] B. M. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th ACM Symposium on Principles of Distributed Computing*, Aug 1988.

[38] 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.

[39] D. Powell, D. Seaton, G. Bonn, P. Verissimo, and F. Waeselynk. The delta-4 approach to dependability in open distributed computing systems. In *Digest of Papers, The 18th International Symposium on Fault-Tolerant Computing*, Tokyo, Japan, Jun 1988.

[40] A. Ricciardi and K. Birman. Using process groups to implement failure detection in asynchronous environments. Technical Report TR 91-1188, Dept of Computer Science, Cornell University, Feb 1991.

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

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

[21] M. Herlihy. Extending multiversion time-stamping protocols to exploit type information. *IEEE Transactions on Computers*, C-36(4):443–448, Apr 1987.

[22] 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.

[23] D. Johnson and W. Zwaenopoel. Sender based message logging. In *Seventeenth International Symposium on Fault-Tolerant Computing*, pages 14–19, Pittsburgh, PA, Jun 1987.

[24] D. Johnson and W. Zwaenopoel. Recovery in distributed systems using optimistic message logging and checkpointing. In *Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 171–181, Toronto, Canada, Aug 1988.

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

[26] 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.

[27] H. Kopetz, G. Grunsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In *International Working Conference on Dependable Computing for Critical Applications*, pages 167–174, Santa Barbara, California, Aug 1989.

[28] H. Kopetz and W. Merker. The architecture of mars. In *15th Annual Symposium on Fault-Tolerant Computing*, pages 274–279, Ann Arbor, Michigan, Jun 1985.

[29] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. Lazy replication: Exploiting the semantics of distributed services. Technical Report MIT/LCS/TR-484, MIT Laboratory for Computer Science, Cambridge, MA, Jul 1990.

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

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

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

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

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

[12] F. Cristian, B. Dancey, and J. Dehn. Fault-tolerance in the advanced automation system. Technical Report Research Report RJ 7424, IBM Almaden Research Center, Apr 1990.

[13] D. Daniels and A. Z. Spector. An algorithm for replicated directories. In *Second Annual ACM Symposium on Principles of Distributed Computing*, pages 104–113, Montreal, Canada, Dec 1983.

[14] D. Davcev and W. A. Burkhard. Consistency and recovery control for replicated files. In *Tenth ACM Symposium on Operating System Principles*, pages 86–96, Orcas Island, WA, Dec 1985.

[15] D. Dolev, J. Y. Halpern, and R. Strong. On the possibility and impossibility of achieving clock synchronization. In *Proceedings of 16th Anual ACM STOC*, pages 504–511, Washington D.C., Apr 1984.

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

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

[18] D. K. Gifford. Weighted voting for replicated data. In *Proceedings of the Seventh Symposium on Operating Systems Principles*, pages 150–162, Dec 1979.

[19] J. Gray. Notes on database operating systems. In *Lecture Notes in Computer Science 60*, pages 393–481. Springer-Verlag, Berlin, 1987.

[20] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Transactions on Computer Systems*, 4(1):32–53, Feb 1986.

dependent ordering, membership, and recovery. Second, it suggests a decomposition of a fault-tolerant distributed system into a set of primitive building blocks. Decomposing a system into its fundamental building blocks has three advantages: (1) it simplifies the process of implementing, debugging, and optimizing each piece; (2) it aids in finding the inherent dependencies among the pieces, and as a consequence, reduces unnecessary interference among them; and (3) it allows the application to select exactly the right combination of building blocks to provide the required functionality. We believe that the importance of these advantages will increase as fault-tolerant systems become more prevelant in critical applications.

References

[1] P. A. Alsberg and J. D. Day. A principle for resilient sharing of distributed resources. In *Proceedings of 2nd International Conference on Software Engineering*, pages 627–644, Oct 1976.

[2] K. Birman. Replication and fault-tolerance in the ISIS system. In *Tenth ACM Symposium on Operating System Principles*, pages 79–86, Orcas Island, WA, Dec 1985.

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

[4] K. Birman, T. Joseph, T. Raeuchle, and A. Abbadi. Implementing fault-tolerant distributed objects. *IEEE Transactions on Software Engineering*, SE-11(6):502–508, Jun 1985.

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

[6] K. Birman, A. Schiper, and P. Stephenson. Lightweight causal and atomic group multicast. Technical Report 91-1192, Department of Computer Science, Cornell University, Feb 1991.

[7] A. Birrell, R. Levin, R. Needham, and M. Schroeder. Grapevine: An exercise in distributed computing. *Communications of the ACM*, 25(4):260–274, Apr 1982.

[8] S. A. Bruso. A failure detection and notification protocol for distributed computing systems. In *Proceedings of the IEEE 5th International Conference on Distributed Computing Systems*, pages 116–123, Denver, CO, May 1985.

computer systems. Because these systems use synchronized clocks, the algorithms for various protocols in these systems cannot make use of the partial order among various events in the distributed system and hence they resort to more expensive total order. On the other hand, in Consul, partial order has been used to provide more efficient algorithms for these protocols, but no real time guarantees are made.

Isis and Delta-4 do not make use of the synchronized clocks for implementing various fault-tolerant protocols. The Delta-4 project seeks to define a dependable distributed, real-time operating system that allows integration of heterogeneous computing elements, while Isis is a distributed programming environment that provides tools for building fault-tolerant applications. Both of these systems provide causal ordering but they do not preserve the context graph and present it to the application. As a result, these systems cannot provide fault-tolerant algorithms that make use of the communication history of the system. In particular, weaker orderings such as semantic dependent ordering, that make use of the communication history, cannot be implemented in these systems.

A final advantage of Consul is that it provides a configurable architecture in which an application designer can build a system around a given collection of protocols with minimum effort. As a result, the system can satisfy the diverse needs of many different applications with little overhead and in a way that forces an application to pay for only the functionality that it needs. The use of reconfiguration protocols makes it easy to modify the system architecture or add new protocols to the substrate without affecting existing components.

8 Conclusions

This paper describes the design of Consul, a communication substrate for fault-tolerant, distributed programs. Consul consists of a suite of fault-tolerant communication protocols that together provide various fault-tolerant services such as broadcast, membership, and recovery. These protocols together form a substrate that can be used to build fault-tolerant applications. Specifically, the fault-tolerance support includes process failure detection, restart of failed processes, and reliable communication between processes. The support for distributed processing includes interprocess communication within a group of processes and different kinds of orderings among messages exchanged in the system.

This paper makes two major contributions. First, it gives novel algorithms for exploiting the partial ordering among the messages exchanged in the system. In particular, algorithms are given for semantic

view. However, in all of these approaches, the complete protocol has to be restarted when a process fails while the protocol is in progress. On the other hand, our protocol manages such failures differently—failures or recoveries detected while the protocol is in progress are taken into account incrementally by updating **SuspectUpList** or **SuspectDownList** appropriately. Moreover, the protocol proposed in [40] only establishes a consistent time when a failed process is to be removed. In particular, it assumes that detecting and establishing the failure of a process is implemented elsewhere. Since in asynchronous systems it is impossible to distinguish with certainty between a failed processor and one that is merely slow, the best that can be done is reaching a tentative conclusion about a process that is suspected to have failed. Such a conclusion is reached by using some heuristics that typically involve communication among all the processes, for example, by using ack and nack messages, as we do above.

Another advantage of our protocol relative to the other approaches is that it relaxes the requirement that removal of a failed process from the membership list be totally ordered with respect to all other events. In particular, a process waits to update its membership list only until it has determined the last message sent by the failed process; it need not wait for other processes to update their membership lists. In contrast, other protocols force a process to wait until all functioning processes have confirmed the failure.

A final advantage is that removal of failed processes from the membership list need not be done at the same time at all the processes. This results from the fact that sf-groups are created dynamically at each process, and these groups need not be the same at all processes. Thus, a process that does not have to merge two sf-groups will be able to remove the members of the first group before another process that does have to merge the two groups. This improves the efficiency of the application and simplifies the design of the protocol. In contrast, other protocols wait until all the processes have formed their sf-groups before removing the failed process.

7.3 Fault-tolerant Systems

We compare the system design of Consul with some of the recent fault-tolerant systems being developed. These include MARS [28, 26], AAS [12], DELTA-4 [39], and ISIS [6]. Both MARS and AAS are distributed real-time systems that employ synchronized clocks to implement various fault-tolerant services provided by the system. MARS is a system designed for distributed real-time process control applications, while AAS is designed to replace the present en-route and terminal approach U.S. air traffic control

In the second category, the semantics of the application have been exploited to come up with a solution. In [13], semantic information has been used to implement a replicated directory, while in [14], the authors use semantic information to implement replicated files. Our approach differs from these two in that we maximize the concurrency by dividing different operations into op-groups, and our approach generalizes easily beyond files and directories. In [21], semantic information is used to efficiently implement multiversion timestamping protocol for atomic transactions. While this work is similar to ours, the two approaches differ in two aspects. First, we efficiently implement operations on an object instead of atomic transactions. Second, we deal with objects replicated over multiple sites. That is, our emphasis is on increasing concurrency of independent operations over multiple sites rather than increasing concurrency among transactions on a single site.

Finally, we compare our work with [29]. Here, lazy replication has been proposed as a way to preserve consistency by exploiting the semantics of the service's operations to relax the constraints on ordering. Three kinds of operations are supported: operations for which the clients define the required order dynamically during the execution, operations for which the service defines the order, and operations that must be globally ordered with respect to both client-ordered and service-ordered operations. While this approach is best suited for client-defined ordering, many applications do not fit in this model. Rather, they involve a collection of different kinds of operations, some requiring total ordering and some requiring partial ordering with respect to one another. Our approach performs much better in a case such as this when mixture of these different operations needs to be applied to an object. The approach proposed in [29] must resort to a total ordering in such a case.

7.2 Membership

Membership protocols have been proposed for both synchronous and asynchronous environment. Membership protocols in synchronous systems include [10, 16, 27, 33, 43]. All these protocols make use of synchronized clocks to maintain a consistent view of which processes are functioning at every clock tick.

Because the concepts of asynchrony and membership are incompatible, the membership problem is more difficult in asynchronous systems than in synchronous systems. The protocols proposed in [3, 8, 9, 35, 40] and the one proposed in this paper assume an asynchronous environment. The protocol proposed in [8] communicates failure information by diffusion. This algorithm, however, does not attempt to maintain a consensus view of the configuration. The protocols proposed in [3, 9, 40] do maintain a consistent

Ops/sec	Checkpoint interval (in sec)	Time for 100 Ops (in sec)
10	No Checkpointing	10.0
10	5.0	10.0
10	2.0	10.0
10	1.0	10.01
20	No Checkpointing	5.99
20	5.0	6.0
20	2.0	6.035
20	1.0	6.07
40	No Checkpointing	4.0
40	5.0	4.05
40	2.0	4.15
40	1.0	4.25
50	No Checkpointing	2.0
50	5.0	2.1
50	2.0	2.3
50	1.0	2.35

Table 4: Measure of Checkpointing Overheads

checkpoint intervals of 5.0 sec or higher for the observed operation rates.

7 Related Work

Considerable attention has been given to the design of various fault-tolerant protocols and systems. In this section, we compare our work with some of the recent work being done in this area. As explained before, we have presented novel algorithms for semantic dependent ordering and membership in Consul, and therefore, in the following we compare these two functions to the related approaches in the literature. Finally, we compare the overall system design of Consul with some other fault-tolerant systems that also provide these fault-tolerant services.

7.1 Message Ordering

Our approach in managing replicated objects is similar in many respects to approaches taken elsewhere. These approaches may be classified into two categories. The first category includes those protocols where the semantics of the operations are not exploited and a total order is imposed to implement replicated objects or a related constructs. Examples of this approach include [1, 4, 2, 7, 18, 20, 32, 37, 36]

Failure Detection protocol needs to receive every message exchanged in the system. Thus, the Divider protocol demultiplexes every message to the Failure Detection protocol and the Order protocol.

6.4 Checkpointing Overhead

An experiment to measure the checkpointing overhead is done as follows. Two clients at different processors issue operations at a known, fixed rate, and the elapsed time for every 100 operations is measured. In the experiment, 5,000 operations were issued by each client. Note that the time measured in this experiment cannot be used to calculate the response time since it is dominated by the time taken to issue the operations. However, since the operations are issued at a fixed rate, the effect of the checkpointing overhead is uniform over all the operations. This experiment includes both the asynchronous message logging done by Psync and the checkpointing done by the order protocol. For the former, a given message's identifier, dependencies, and contents are stored. For the latter, the message identifiers of all the messages in the participant's view are checkpointed; this is done atomically when a wave becomes complete.

The time measured is for four different rates of operations issued by the clients. For each rate, the elapsed time for 100 operations is measured under different checkpointing intervals. The results are shown in Table 4. All the operations issued were commutative operations, with the semantic dependent ordering protocol being used throughout.

Two observations can be made from these measurements. First, the checkpointing overhead increases with the increase in the rate at which clients issue operations. For example, the overhead is 0.35 sec per 100 operations when the clients issue operations at 50 ops/sec and the checkpointing interval is 1 sec, while the overhead is 0.01 sec per 100 operations when operations are issued at 10 ops/sec. The reason for this is that the system performs fewer operations per unit time when the rate of issue of operations is lower, leading to more idle time to do the checkpointing. As a result, its effect on the time to process an operation is less.

The second observation is that the overhead of checkpointing increases as the checkpoint interval is reduced. Thus, this overhead is 0.1 sec per 100 operations when operations are issued at 50 ops/sec and the checkpointing is done every 5 sec, while this overhead increases to 0.35 sec per 100 operations when the logging is done every 1 sec. This is expected, as the time spent to do the checkpointing per unit time increases as the checkpoint interval is reduced. The effect of checkpointing is almost negligible for

and 4-replica systems. These results are shown in Table 2.

% of comm. operations	Semantic Dep. Order	Total Order
0	3.7	3.6
50	3.55	3.6
75	3.2	3.6
90	2.9	3.6
99	2.7	3.6
100	2.7	3.6

Table 1: System Response Time (in msec) for a 2-replica system

% of comm.	Semantic Dep. Order		Total Order	
operations	3-replica	4-replica	3-replica	4-replica
0	4.1	4.45	4.0	4.3
100	2.75	2.8	4.0	4.3

Table 2: System Response Time (in msec) for a 3-replica and 4-replica system

6.3 Membership and Recovery Protocols

The overhead of the failure handling protocols in the absence of failures is measured by extending the configuration to include the Membership, Failure Detection, and Recovery protocols. The code size for these protocols is only about 1500 lines, mainly because of the inherent support Psync provides for such activity by maintaining a replicated context graph. Once again, none of the protocols do any logging or checkpointing in this experiment. The response time was measured in the same way as described above for various mixes of commutative and noncommutative operations. The results are shown in Table 3.

The overhead imposed by the failure handling protocols is about 0.6 msec per operation. This overhead is due to two factors. First, since the communication substrate includes more protocols, the Divider protocol has a larger set of protocols to which to demultiplex incoming messages. Second, the

% of comm. operations	Response Time
0	4.2
50	4.1
75	3.8
90	3.6
99	3.3
100	3.2

Table 3: Response Time with Failure Handling Protocols (in mesc)

6.1 Psync Timings

To measure the performance of Psync, one byte messages were exchanged between a pair of user processes directly on top of Psync. In this test, the resulting average round trip delay was measured as 2.9 msec. This number is derived by exchanging messages for 10,000 trips (20,000 total messages) and reporting the elapsed time for every 1,000 round trips. Each of these measurements was then divided by 1,000 to produce the average.

6.2 Performance Using Semantic Dependent Ordering

To determine how well the semantic dependent ordering protocol performs, we compared the performance of the replicated directory using the semantic dependent protocol with the same application using a total ordering protocol. In this experiment, we focused on measuring the average *response time* of the system, i.e., the elapsed time between the time an operation is issued by the client and the time that operation is applied to the local copy of the directory. The time needed to actually perform the operation is not included.

For this experiment, the communication substrate was configured to include Psync, the Divider protocol, and the appropriate order protocol. There was no logging or checkpointing done by any of these protocols. The system was configured to run on two, three and four processors respectively. In the case of the semantic dependent ordering protocol, the average response time depends heavily on the overall mix of the commutative and the noncommutative operations, so the mix was varied across different runs. In each case, the response time is derived by having clients on each processor apply 10,000 operations, with a varying percentage of commutative operations uniformly distributed, and reporting the elapsed time for every 1,000 operations. Each of these measurements was then divided by 1,000 to produce the average response time.

The results for the system configured for two replicas are shown in Table 1. As expected, the semantic dependent ordering protocol improves the response time of the system as the percentage of commutative operations increase. The response time is 2.7 msec when all the operations applied are commutative, giving an improvement of about 25% over the use of a total ordering protocol. Another important point to note is that the semantic dependent ordering protocol performs almost as well as the total ordering protocol when all the operations are noncommutative. That is, the overhead of the protocol is negligible, leading to minimal effect on system performance. Similar improvement was observed for the 3-replica

normally, *even when the process referred to in the message is itself.* In other words, the membership protocol masks *itself* out of the participant set when it receives the ⟨**p is down**⟩ message, and then back in when it receives the ⟨**p is up**⟩ message. This is necessary so that p will make correct stability decisions for those operations that occurred while it was down. This also implies that the mask on the participant set must be saved when a checkpoint is performed so that it is correctly restored upon recovery.

Finally, notice that by doing frequent checkpoints, the recovering site is able to minimize the size of the second portion of the context graph, thereby reducing the amount of duplicate work it does upon recovery. However, this checkpointing is actually unnecessary: the site could *replay* the entire communication history in order to restore the state of the object, as is done in other systems [23, 24].

6 Performance

Consul has been completely implemented. The code consists of approximately 10,000 lines of C code, of which 3,500 is Psync. Consul is implemented in the *x*-Kernel, an operating system kernel designed explicitly for experimenting with communication protocols [22].

We have built two different applications using Consul: the replicated directory described in Section 3 and a distributed word game. Both of these applications have been tested under varying configurations for two, three and four replicas. These configurations differ in several ways. One is the type of ordering protocol used; some use semantic dependent ordering, while others use total ordering. Another is whether or not they contain various failure handling protocols. A third is whether checkpointing is performed and at what interval. Our experience has been that it is easy to move from one configuration to another without any modifications to the substrate.

This section reports on the performance of various protocols in Consul and the overheads they impose on the overall performance of the system. All of the numbers reported here have been taken from the replicated directory application running on a collection of diskless Sun 3/75 workstations connected by a lightly loaded 10Mbs Ethernet; stable storage is simulated by a process executing on another Sun 3/75. Various experiments were designed to measure the performance of Psync and the semantic dependent ordering protocol, as well as the overhead of the various failure handling protocols.

The reconstruction of the context graph is done by Psync in two steps. Recall that Psync logs messages onto the stable store at regular intervals. In the first step, these messages are retrieved and placed into the new image of the graph. However, since the logging of messages and the checkpointing of the view by the order protocol are done independently, the logged messages may not be sufficient to reconstruct the complete context graph down to *wave(n-view)*. In such a case, a second step is invoked that retrieves the the missing messages from other functioning processes. Since all the operations received during this stage have already been applied by the application, the other protocols in the communication substrate are quiescent during this stage.

The second and third stages of the recovery service are initiated simultaneously by the Recovery protocol at the recovering site by transmission of the **restart** message. The second stage deals with updating the state of the process to the current state of the system. On receiving the **restart** message, a functioning process retransmits all missing messages to the recovering process. The recovering process processes these messages as it would normally, but with two exceptions. First, all the protocols function in a passive mode, that is, they do not send any messages during this time. The intuition is that, although the recovering site is replaying the past, it was not an active participant in the decisions and hence should not send messages. Second, the recovering site does not accept new operation requests from client programs. Such messages are associated with normal processing, and so are deferred until the recovery phase is complete.

The third stage deals with incorporating the process into the membership list and determining when the recovering process can start participating actively in the system. This is done by an asynchronous notification on receipt of the **restart** message. Intutively, the recovering process starts participating actively in the system after it has been incorporated into the membership list by every alive process, thus ensuring that it has received all messages prior to this time. Recall that every alive process sends an $\langle$**Ack, p is up**$\rangle$ message before incorporating the process in the membership list. Thus, the recovering process determines that it has been incorporated by all the alive processes when it has received $\langle$**Ack, p is up**$\rangle$ message from every such process. The Recovery protocol checks for this condition and sets every protocol in active mode when it is satisfied. The completion of this stage completes the recovery service.

The remaining question is how the Recovery protocol knows which processes are functioning so that it can determine when all $\langle$**Ack, p is up**$\rangle$ have been received. To determine this, the recovering site processes messages associated with the failure handling protocols during the passive phase as it would

As shown in Figure 9, the graph can be divided into four regions. Define *wave(n)* to be the wave containing node *n*. The first portion is from the root of the graph down to and including the nodes in *wave(n-view)*, where *n-view* is any node in the newly restored view. Any operations in this portion have already been applied. The second portion is from below the restored view down to and including the nodes in *wave(n-failed)*, where *n-failed* is the node corresponding to the ⟨**p is down**⟩ message generated by the process that detected the failure of *p*.[3] Any operations in this region may or may not have been applied. Similarly, the third portion is from below *wave(n-failed)* down to and including the nodes in *wave(n-restart)*, where *n-restart* is the node corresponding to the ⟨**p is up**⟩ message generated when *p* recovers. Operations in this area were missed due to the failure of *p*. The fourth and final portion of graph consists of those nodes below *wave(n-restart)*. Any operations here are new operations that *p* will apply once recovery is complete.

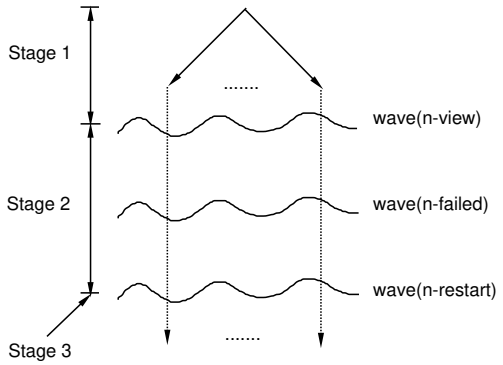

Figure 9: Context Graph at Recovery

It is clear from the above discussion that the operations that the recovering process needs to apply before resuming normal processing are those in the second and third portions. That is, it should apply those operations in the graph from below *wave(n-view)* up to and including *wave(n-restart)*. This is done in the various stages of recovery.

As mentioned above, the state of the substrate is restored to the checkpointed state in the first stage. This corresponds to restoring the context graph until *wave(n-view)* in Figure 9. Here, the (Re)Start protocol first collects the checkpoints from all the protocols and restarts each based on this information.

[3]There may be more than one such message if the failure was detected simultaneously by multiple processes. These messages will be siblings in the context graph, however, so the portion of the graph defined above is the same.

graph have been applied as well. The Failure Detection, Membership, and Divider protocols are stateless and so do not perform any checkpointing. The (Re)Start protocol saves information to recreate the proper connections among various protocols used by the application following a crash. In general, the design of the recovery service is such that every protocol saves enough information to recover after a failure and this recovery of the protocol is initiated by the Recovery protocol. The application is checkpointed separately in an operation that is coordinated with the checkpointing of the view.

5.2 Stages of Recovery

Given this checkpointing and message logging strategy, the recovery service goes through three stages as shown in Figure 8. The first stage of this process restores the substrate to the checkpointed state, the second stage restores the substrate and the application to the current state of the system, and the third stage initiates the Membership protocol to incorporate the process into the membership list. Since the fact that the context graph encapsulates the entire communication history of the system makes it the key aspect of the recovery service, we outline these stages by focusing on the context graph.

Stage	**Actions at the recovering site**
Stage 1	(Re)start: Collect checkpoints from all protocols.
	(Re)start: Restart each protocol with required parameters.
	Psync: Reconstruct context graph from stable store.
	Psync: If context graph reconstruction is incomplete, send retransmit request for missing messages.
Stage 2	Recovery: Set every protocol in passive mode.
	Recovery: Send **restart** message.
	All protocols function passively.
Stage 3	Recovery: If every alive process has acknowledged ⟨**p is up**⟩ message, set protocols to active mode.

Figure 8: Stages of Recovery Service

In summary, we have shown the following. First, since a process starts accepting messages from a recovering process as soon as the recovering process is added to the membership list, by Theorem 6 we conclude that every functioning process starts accepting messages from a rejoining process at the same time in the view. Theorem 9 confirms that the decision reached by the membership protocol about a failed (or suspected failed) process is consistent at all functioning processes. Thus, first two conditions of Section 4.1 are satisfied. The last two conditions are ensured by Theorems 5, 7 and 8. Finally, Theorem 10 guarantees that the order of the removal of failed processes is correct.

5 Recovery

The recovery service is concerned with restoring the state of a failed process in such a way that the system state remains consistent. In Consul, the recovery service is implemented primarily by the Recovery and Stable Storage protocols, although other protocols are involved in the recovery process. When a processor recovers, three things must happen before it can resume normal processing. First, an appropriate state of the communication substrate must be reconstructed, including the states of all the protocols, the local image of the Psync context graph, and the process' view of the conversation. Second, processes at other sites must be informed so that the Membership protocol described in the previous section can be initiated. Third, the actual state of the application must be brought up-to-date. We describe how these tasks are accomplished by outlining the sequence of events at a process that fails and subsequently recovers.

5.1 Checkpointing

The recovery service in Consul requires every protocol to do its own checkpointing during normal operation. These checkpoints include various parameters needed to recover the state of the protocol following a failure. An important part of the state of the substrate is the image of the Psync context graph and the view of the participant. To restore this, Psync checkpoints the participant set and the conversation identifier (**cid**). Psync also logs parts of the context graph periodically onto stable storage to minimize the message retransmission performed when the context graph image is rebuilt.

In addition, the Order protocol checkpoints the view of the local process; a value that consists of a set of message identifiers. This checkpointing is done by the Order protocol only after all the operations in the view have been successfully applied; this guarantees that all preceding operations in the context

Since $r \in L_p$, p has received every message from r. Since p has not received any of the NACKs, r must have sent $\langle$**Ack, q is down**$\rangle$ for every $\langle$**q is down**$\rangle$ message where $q \in Q$. Thus, we have the following:

[B] : r has sent $\langle$**Ack, q is down**$\rangle$ for every $\langle$**q is down**$\rangle$ message for all $q \in Q$.

Since p has not received any of these NACKs and has received all of the $\langle$**q is down**$\rangle$ messages, none of these NACKs are followed by a $\langle$**q is down**$\rangle$ message. So, we conclude the following:

[C] : For any $q \in Q$, the message $\langle$**Nack, q is down**$\rangle$ is not followed by any $\langle$**p is down**$\rangle$ message where $p \in Q$.

Let s be some arbitrary process in ML. We say that "q is NACKed by s" if process r has received the message $\langle$**Nack, q is down**$\rangle$ from s, and this is the first NACK received immediately following the $\langle$**q is down**$\rangle$ message. Also, we say "q is freed before s" if q is NACKed before s; i.e., the first NACK message immediately following $\langle$**q is down**$\rangle$ is received before the first NACK message immediately following $\langle$**s is down**$\rangle$. Furthermore, let q be NACKed by s, where $s, q \in Q$. From [C], we see that $\langle$**Nack, q is down**$\rangle$ is not followed by $\langle$**s is down**$\rangle$ message. Since $\langle$**Nack, q is down**$\rangle$ is sent by s and [B] implies that r sent $\langle$**Ack, s is down**$\rangle$ in response to $\langle$**s is down**$\rangle$, r must have received the $\langle$**s is down**$\rangle$ message before receiving $\langle$**Nack, q is down**$\rangle$. Since r received the $\langle$**s is down**$\rangle$ message and later received $\langle$**Nack, q is down**$\rangle$ from s, s must have been freed before r received $\langle$**Nack, q is down**$\rangle$. This implies that s has been freed before q. So, we have the following:

[D] : If a process q_i is NACKed by another process q_j, then q_j is freed before q_i for all $q_i, q_j \in Q$.

From [A], every process in Q must be NACKed by some other process in Q, and from [D] we see that every process in Q has a process in Q that was freed before it. Since Q is a finite set and "freed before" is a total relation, this is impossible for any finite non-empty Q. Thus, the set Q must be empty. Therefore, $Q = L_r - L_p = \phi$. Thus $L_r = L_p$. So, $q \in L_r \Leftrightarrow q \in L_p$. □

Theorem 10 *The removal of failed processes from the membership list follows the total order of sf-groups and all processes in one sf-group are removed simultaneously.*

Proof: By Theorems 4-9. □

it. However, a process that was down at that time may not have received it. All the messages that are in the context graph of an alive process and that are missing in the context graph of the recovering process are retransmitted. Since message m is stable, it is retransmitted if it is not in the context graph of the recovering process. A recovering process starts participating in the conversation only after the retransmission of all the missing messages is complete, in particular after the receipt of message m. Thus, a failed process that rejoins the membership list has received all the messages that are declared stable. □

Before considering the next theorem, we observe that we can always consider the logical time at which a given membership check period started as the same in all alive processes. This follows from the property that consecutive membership check periods (or equivalently, sf-groups) can be combined into a single membership check period without changing the semantics. Thus, given a specific check period, we can successively merge earlier check periods as required in each process to obtain a single membership check period that starts at the same logical time everywhere. This iteration is guaranteed to stop since the initial membership check period must start at the same time in every process. Note that this merging is done only for the purposes of the proof, and is not actually reflected in the protocol itself.

In the following theorem we prove that if one process decides that a process p has failed, every alive process eventually decides the same.

Theorem 9 *Let L_p and L_r be the set of alive processes at the end of membership check period containing* $\langle$**q is down**$\rangle$ *in views of processes p and r respectively. If $r \in L_p$ and $p \in L_r$ then $q \in L_p \Leftrightarrow q \in L_r$.*

Proof: Using the above observation, we logically combine sufficient previous membership check periods so that the periods in the two processes start at the same time. Let $Q = L_r - L_p = \{q_1, ..., q_k\}$. That is, Q is the set of processes that are in membership list of process r but not p at the end of the membership check period. Assume that $q \in Q$. Since q is in the membership list of r, the message $\langle$**q is down**$\rangle$ is immediately followed by $\langle$**Nack, q is down**$\rangle$ from some process (say s) in the view of process r. Since q is not in the membership list of p, message $\langle$**q is down**$\rangle$ is not followed by any $\langle$**Nack, q is down**$\rangle$ message in p's view. Since p receives every message from all processes in L_p, and has not received the $\langle$**Nack, q is down**$\rangle$ message from s, then s must belong to Q. Thus, we have the following:

> **[A] :** For every $q \in Q$, the $\langle$**q is down**$\rangle$ message is followed by a $\langle$**Nack, q is down**$\rangle$ message from some process $q_i \in Q$ in the view of r but not in the view of p.

Theorem 7 *Stability and completeness decisions taken in the application during the execution of membership protocol are valid.*

Proof: Stability decisions are correct if a message is decided stable when it is followed by messages from every alive process. The stability check of a message *m* can go wrong if some process that has a message *m'* following *m* is not considered for the stability check and *m'* has not been received. If a process *p* fails, then it is removed from the membership list at its RDT. Thus, the failed process is removed from consideration for stability checks only after its RDT. Since all the messages from a failed process have been received by its RDT (Theorem 4), all of its messages have been received before it is removed from consideration for stability checks. Thus, removal of a failed process does not introduce any inconsistency in the stability checks. When a process *p* recovers, it starts sending messages only after its RUT. Since the recovering process is incorporated into the membership list at its RUT, it is considered for stability checks for all the messages at or following RUT. Thus, stability checks remain consistent during the recovery of a process.

Completeness decisions are correct if a wave is decided to be complete only when it contains all of its messages. As noted in Section 2, this is typically done by testing the stability of a message in the wave. In the presence of failures, however, a failed process is not considered to determine stability, and a wave may contain messages from processes that later fail. This does not cause a problem because a failed process is not considered when determining the completeness of a wave only after it has been removed from the membership list; i.e., after its RDT. But by Theorem 4, all messages from failed processes have been received by its RDT. Therefore, a wave that is determined to be complete has all messages from failed processes and all the alive processes (since all the alive processes are considered to determine stability), and as a consequence, removing a failed process does not cause any inconsistency in wave completeness checks. A recovering process *p* starts sending messages after its RUT, so no waves between its RDT and RUT inclusive contain a message from *p*. Since *p* is added to the membership list at its RUT, it is considered for completeness checks for all succeeding waves. Thus, completeness decisions are consistent during the process of process recovery. □

Theorem 8 *Alive processes have received all stable messages.*

Proof: A message *m* is declared stable when there is a message from all processes in ML following *m* in the context graph. Thus, all the processes that were alive at the time *m* was declared stable have received

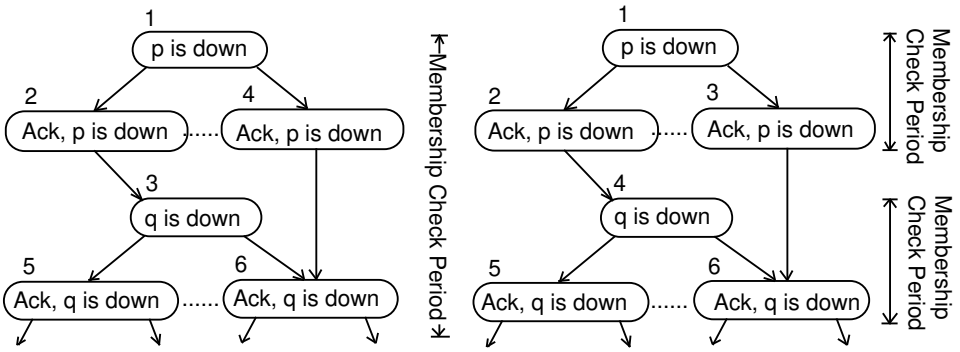

Figure 7: Two Different Processes' View

down⟩ from every alive process. Thus, all messages sent by p that have been received by any alive process prior to sending ⟨**Ack, p is down**⟩ have been received by q. Since a process ignores any further incoming messages from p as soon as it sends ⟨**Ack, p is down**⟩ q receives no other messages from p. Thus, q has received all messages from p at its RDT; i.e., when it decides that p is down. □

Theorem 5 *An alive process receives all the messages in the conversation.*

Proof: The membership check period ends at the RDT or RUT of any process. From Theorem 4, we know that p has received all messages from failed process. Since the membership check period terminates at a wave that contains messages from all alive processes' (**ACK** or **NACK**), all messages from all alive processes have also been received. Thus, a processes receives all messages in the conversation in the membership check period. Since Psync guarantees the delivery of all the messages in the normal state (i.e., when a process is not in a membership check state), an alive process receives all the messages in the conversation. □

Theorem 6 *A recovering process is incorporated in the conversation at the same logical time at all alive processes.*

Proof: A recovering process p is incorporated into the membership list when the ⟨**p is up**⟩ message is immediately followed by a ⟨**Ack, p is up**⟩ message from every alive process, i.e., when the wave containing the message ⟨**p is up**⟩ is complete. Since this wave is the same at all processes, p is added to the membership list at the same time by all processes. □

Message	Actions of the Membership Protocol
⟨**p is down**⟩	If a message from p at the same logical time as ⟨**p is down**⟩ has been received, then send ⟨**Nack, p is down**⟩; otherwise send ⟨**Ack, p is down**⟩, stop accepting messages from p, insert ⟨**p is down**⟩ in **SuspectDownList**, and adjust **count**.
⟨**Nack, p is down**⟩	Remove ⟨**p is down**⟩ message from **SuspectDownList** and start accepting messages from p.
⟨**Ack, p is down**⟩	If message ⟨**p is down**⟩ is stable then decrement **count**.
⟨**p is up**⟩	Send ⟨**Ack, p is up**⟩, insert ⟨**p is up**⟩ in the **SuspectUpList**, and increment **count**.
⟨**Ack, p is up**⟩	If ⟨**p is up**⟩ is stable, then incorporate p into the membership list, remove ⟨**p is up**⟩ from **SuspectUpList**, and adjust **count**.

Figure 6: Membership Protocol

left receives ⟨**q is down**⟩ before the final ⟨**Ack, p is down**⟩ message.[2] This is possible because there is no path from the final ⟨**Ack, p is down**⟩ to the ⟨**q is down**⟩ message. Thus, both p and q are considered in one membership check period. In contrast, the process on the right receives all ⟨**Ack, p is down**⟩ messages before receiving ⟨**q is down**⟩. As a result, the right-most process uses two membership check periods. The implication of this situation is that p is removed from the membership list earlier at the right process than in at the left process, thus allowing its execution to proceed without unnecessary delay.

4.3 Correctness Arguments

We now argue that the correctness conditions of Section 4.1 are satisfied.

Theorem 4 *All messages from process p have been received by a process q before q decides the failure of p.*

Proof: The set of messages from process p is the union of all messages from p received at all alive processes. At the RDT of p, the message ⟨**p is down**⟩ is immediately followed by message ⟨**Ack, p is**

[2]Keep in mind that any pair of messages in the context graph for which there is no path leading from one to the other could have been received by the process in either order, and in fact, in different orders at different processes.

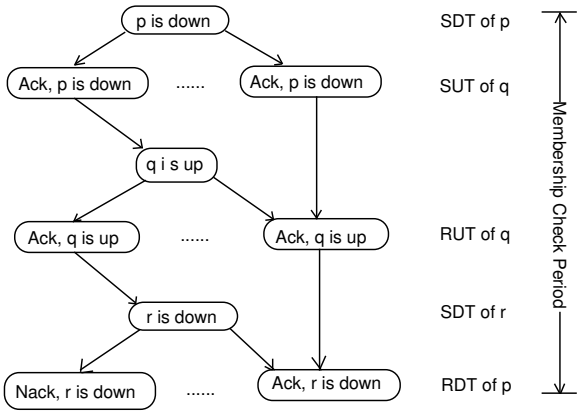

Figure 5: View Representing a Membership Check Period

the ⟨**p is down**⟩ message. A suspected up process is removed from **SuspectUpList** and added to the membership list as soon as the appropriate ⟨**p is up**⟩ message becomes stable. The membership check period ends when all messages in **SuspectDownList** become stable and **SuspectUpList** becomes empty. At this point, all of the suspect down processes are removed from the membership list, and **SuspectDownList** is reinitialized to empty.

Figure 5 illustrates one possible scenario, in which processes p and r are checked for a possible failure and process q rejoins the membership list. The membership check period starts with the arrival of the ⟨**p is down**⟩ message. At the end of this period—that is, the RDT of p—process p is removed from the membership list. However, r remains since ⟨r **is down**⟩ is immediately followed by ⟨**Nack,** r **is down**⟩.

Figure 6 gives a more precise description of the protocol based on actions taken by a given process upon receipt of each type of message. The phrase "**count** is adjusted" means that the variable **count** is assigned the number of messages in **SuspectUpList** plus the number of unstable messages in **SuspectDownList**. In addition, if **count** goes to zero as the result of processing a message, then **SuspectDownList** is reinitialized to empty and the corresponding processes are removed from the membership list.

The key observation about the algorithm is that different processes can form different sf-groups, or equivalently, different processes can have different membership check periods. The reason is that the membership protocol includes a process p for consideration as soon as it receives ⟨**p is up**⟩ or ⟨**p is down**⟩ protocol initiation message. To see the effect of this, consider the scenario outlined in Figure 7, where the numbers above the nodes represent the order in which messages are received. The process on the

Suspected Down Time (SDT): The SDT of a failed process p is the logical time containing the ⟨**p is down**⟩ message.

Actual Down Time (ADT): The ADT of a failed process p is the earliest logical time such that there are no messages from p at or after ADT.

Realized Down Time (RDT): The RDT of a failed process p is the logical time when p is masked out of the membership list.

Suspected Up Time (SUT): The SUT of a process p is the logical time containing the ⟨**p is up**⟩ message.

Realized Up Time (RUT) The RUT of a process ML is the logical time when p is masked back into the membership list.

Furthermore, define a *membership check state* as a state where **SuspectDownList** or **SuspectUpList** is non-empty. In a similar manner, let a *membership check period* be the time interval over which the system is in a membership check state. A membership check period always starts at the SDT or SUT of some process and ends when the two suspect lists are empty. In other words, a membership check period starts when **count** becomes non-zero and continues until **count** becomes zero. The end of a membership check period is always signified by the RDT of some process.

In the protocol, a message is considered stable if it is followed by messages from all the processes that are not in **SuspectDownList.** Note that this definition of stability is applicable only in the membership protocol; all other protocols, including those that run concurrently with the membership protocol, use the standard definition of stability given in Section 2.

4.2.3 Membership Protocol

The main idea of the Membership protocol is to establish agreement among all the alive processes about the membership list at the end of the corresponding membership check period. Thus, if there are n processes of which k are suspected to have failed, agreement on the failure of the k processes is reached if none of the other $n - k$ processes have a message from any of the suspected processes at the same logical time as the first protocol initiation message sent by the detection protocol.

Informally, the membership protocol may be described as follows. Upon receiving a ⟨**p is down**⟩ message, the message is added to **SuspectDownList**. Similarly, upon receiving a ⟨**p is up**⟩ message, the message is added to **SuspectUpList**. The message associated with a suspected down process is subsequently removed from **SuspectDownList** if there is any process that has evidence to contradict the hypothesis that it has failed; that is, if a ⟨**Nack, p is down**⟩ message is received immediately following

or if a running process receives the protocol initiation message for a process in S_2 before receiving all messages associated with the protocol for S_1. Interestingly, since messages are received in partial order, this latter situation can result in different sf-groups being formed at different functioning processes. As discussed more fully in Section 4.2.3, our protocol exploits this fact to allow some processes to remove processes from the membership list earlier than others, while still preserving the semantic correctness of the application.

The other type of membership list modification is the addition of recovering processes. In this case, it is sufficient to add a process to the membership list at every process sometime before the recovered process sends its first message. Once this incorporation is complete, the process participates normally in system activity, including execution of future membership protocol agreement algorithms. Since the set of alive processes must be the same at all the processes while executing, the recovering process must be incorporated at the same logical time with respect to all other Membership protocol events at all processes.

In summary, the order in which membership list modification events are handled is as follows:

- All processes in the same sf-group are removed simultaneously. The order of removal of processes in different sf-groups follows the relative order of the sf-groups.
- A recovering process is incorporated into the membership list at the same logical time at all the processes.

4.2.2 Protocol Preliminaries

For simplicity, we first define the terms and data structures used by the protocol. Message m_2 is said to *immediately follows* m_1 if there is a direct edge from m_1 to m_2 in the context graph; m_2 *follows* m_1 if there is a path from m_1 to m_2 in the context graph. A process p is *suspected down* if it is in the membership list and a $\langle$**p is down**$\rangle$ message has been received. Similarly, a process p is *suspected up* if it is not in the membership list and $\langle$**p is up**$\rangle$ has been received.

The Membership protocol maintains two lists: **SuspectDownList** and **SuspectUpList**. **SuspectDownList** contains the list of $\langle$**p is down**$\rangle$ messages that have been received and **SuspectUpList** contains the list of $\langle$**p is up**$\rangle$ messages that have been received. As described below, messages are removed from these two lists once the process reaches a conclusion about the status of each p. The protocol also maintains an integer variable **count** that contains the total number of messages in **SuspectUpList** plus the number of unstable messages in **SuspectDownList**. Initially the value of **count** is zero and the two lists are empty.

We define the following logical times related to process failures and recoveries, where logical time refers to a wave in the context graph.

the order in which the modifications of the membership list take place. We then describe the actual protocol.

4.2.1 Ordering List Modification Events

Suppose two processes p and q fail at approximately the same time. If the last message sent by p is at the same logical time as the last message sent by q (that is, neither is in the context of the other in the context graph), then p and q can obviously not participate in each other's failure agreement protocol. Since establishing agreement about the failure of a process requires concurrence of all functioning processes, agreement for processes that fail in this way must be done simultaneously. On the other hand, if the last message sent by q is in the context of the last message sent by p, then q may contribute messages to the agreement about p having failed, i.e., q may participate in the failure agreement of p.

Now, expand this scenario to include a third failing process r. Suppose the last message sent by r is at the same logical time as the last message sent by q, but follows the last message sent by p. By the argument made above, this implies that the failure agreement of q and r must also be done simultaneously, leading to the conclusion that all three processes must be treated as a group. In general, then, the failure agreement of a set of processes must be done simultaneously whenever the last message sent by any process in the set is at the same logical time as the last message sent by at least one other process in the set. We formalize this notion by defining a *simultaneous failure group* (*sf-group*) S as follows:

> S is an equivalence class of failed processes under the relation $\leftrightarrow^*$, where $p \leftrightarrow q$ if the last message sent by p is at the same logical time as the last message sent by q and $\leftrightarrow^*$ is the reflexive transitive closure of $\leftrightarrow$.

From the perspective of the Membership protocol at a given process, all processes in an sf-group are treated as a unit: one failure agreement algorithm is used for the entire group and they are eventually removed from the membership list simultaneously. Thus, as the execution of a process proceeds, there are a series of sf-groups totally ordered with respect to one another. Specifically, an sf-group S_2 is said to *follow* another sf-group S_1 if the last message sent by any process in S_2 follows the last message sent by all processes in S_1. In this situation, the failure agreement for S_2 is typically performed after the failure agreement for S_1.

Notice, however, that it is also correct to perform the agreement for S_1 and S_2 simultaneously, i.e., as if they were a single sf-group. This type of merging may be necessary in certain situations, such as if one or more processes in S_2 fail before they can participate in the failure agreement associated with S_1,

4.2 Algorithm

For simplicity, first consider the case where at most one process fails at a time. Assume ML initially contains n processes. The membership protocol is based on the effect that the failure has on the context graph. In particular, since a process obviously sends no messages once it has failed, it can be guaranteed there is no message from the failed process at the same logical time as initiation message sent by the Failure Detection protocol. If, on the other hand, there *is* a message from the suspect process at the same logical time as the initiation message, then it can be viewed as evidence that the process has in fact not failed. In this case, it is likely that the original suspicion of process failure was caused by the process or network being "slow" rather than an actual failure. The Membership protocol uses this heuristic to establish the failure of a process.

The goal of the protocol is to establish an agreement among the $n-1$ alive processes about the failure or recovery of the n^{th} process. The basic strategy is to agree on the failure of the process if and only if none of the $n-1$ processes have received a message from the suspect process at the same logical time as the protocol initiation message. Accordingly, a process sends a positive acknowledgement if it has not received any message from the suspect process at the same logical time as the initiation message. Otherwise, it sends a negative acknowledgement. A functioning process concludes that the suspect process has failed if $n-1$ positive acknowledgements are received in response to the protocol initiation message. In the case of recovery, the process is incorporated into the membership list once all the remaining $n-1$ processes have acknowledged its recovery.

The protocol outlined above does not work in the presence of multiple concurrent failures. Correctly handling simultaneous failures is complex because of the inherent lack of knowledge about the set of processes that participate in the membership agreement process itself. That is, processes may fail or recover at any time and, in particular, they may fail or recover while the Membership protocol is in progress. Another source of complexity stems from the requirement that a consistent order of removal or incorporation of processes in the membership list be maintained. This order must be the same at all the processes to ensure correctness of the application, however, it is not at all clear what this order should be, or even the correct interpretation of "the same."

We first address this latter question by deriving an order in which these list modification events must be performed. We show that the semantics of *remove*—removing a failed process from the membership list ML—and *join*—incorporating a recovering process in the membership list ML—put a restriction on

ordered with respect to other events such as interprocess communication to guarantee that messages are committed consistently, but the failure notification is not necessarily passed on to the application. We note, however, that our protocol can easily be adapted to support a user-level facility by including membership change events in the portion of the context graph seen by the application.

The Membership protocol is based on the partial order provided by Psync. As a result, it requires less synchronization overhead and performs especially well in the presence of multiple failures. In particular, the Membership protocol sits on top of Psync and coordinates the way in which processes modify their local participant list using the Psync primitives **maskin** and **maskout**.

4.1 Correctness Criteria

Psync maintains the context graph and the membership list; the latter is denoted ML. In the presence of failures and recovery, an application may take an inconsistent action because of the changes being made in the membership list and in the context graph. To ensure the correctness of the application implemented over Psync in the presence of failures, Membership must guarantee the following two properties: all functioning processes receive the same set of messages in partial order, and the decisions taken by the application based on these messages are consistent even in the presence of failures. These two properties are satisfied by the following four conditions:

- All functioning processes reach the same decision about a failed (or suspected failed) process.
- Every functioning process starts accepting messages from a recovering process at the same logical time.
- Stability and completeness decisions are made correctly; i.e., a message is considered stable only if it is followed by a message from all the other functioning processes, and a wave is considered complete only when it has all of its messages.
- Every process receives all the messages in the conversation.

The first two conditions imply that every functioning process receives the same set of messages from a failed process and a recovering process, respectively. Hence, they guarantee that the context graphs and membership lists are the same at every processor. The last two conditions ensure that various decisions taken by different processes based on the context graph are consistent while the membership list is being modified.

Since a total ordering is required to execute operations in set S_2, we can execute all the operations from sets $S_2, S_3, ..., S_k$ in the **current_wave** as soon as property $\mathbf{C}_2$ is satisfied and the **current_wave** is complete. Thus, in cases where the state machine has operations that can be subdivided into more than two sets of commutative operations, the partial ordering is used to execute the operations in the first set and a total ordering is used to execute operations not in first set. The net result is that additional concurrency for the first set S_1, but not for the other sets.

4 Membership

Consul's membership service is implemented by a pair of protocols: Membership and Failure Detection. The Membership protocol is used to reach agreement among the currently executing processes that a failure or recovery event has occurred. The task of detecting failure or recovery events is handled by the Failure Detection protocol. Specifically, this protocol monitors the messages exchanged in the system, and upon suspecting a change of state of a process, initiates the membership protocol by submitting a distinguished message to the conversation. In Consul, a failure is typically suspected when no message has been received from a process in some interval of time, while recovery is based on the asynchronous notification generated when the recovering process executes the Psync **restart** primitive. Note, however, that the Failure Detection protocol is independent of the Membership protocol, and may employ any strategy to detect process status changes.

There are actually two different types of membership protocols, each serving different purposes [42]. The first can be viewed as a user-level service that typically translates the failure or recovery confirmation into an event that is then ordered with respect to other events in the system. This ordering is then made available to the application to use in making decisions. Examples of this kind of protocol include [3, 9, 10, 27].

The other type of membership protocol—of which the one presented here is an instance—is sometimes called a *monitor protocol* [42]. In contrast to the user-level orientation of the first type, these protocols are used by the system itself to maintain a consistent view of which processes are functioning and hence participating in system decisions. For example, such information is used in reliable multicast protocols to determine when a message has been received and acknowledged by every functioning process so that it can be committed to the application. The processor failure or recovery event must again be consistently

Proof: This proof follows from the way op-groups are constructed. Since an op-group is constructed by the order in which operations are executed, all the operations in an op-group are executed consecutively. Furthermore, the same deterministic sort algorithm is used on the op-group containing noncommutative operations to determine the order in which to execute these operations. As a result, the execution of operations in an op-group containing noncommutative operations follows the same total order at every process. □

Theorem 3 *The order of op-group execution at every process is the same and this order preserves causality.*

Proof: A commutative operation is executed when all the predecessors of this operations have been executed. This implies that the order of the execution of commutative operations follows causality. A noncommutative operation is executed when it is in the current wave and the continuation property is satisfied. The current wave consists of the earliest wave containing an unexecuted operation. Thus, when noncommutative operations are executed, all their predecessors have also been executed. Thus the order of the execution of noncommutative operations also preserves causality.

Since the execution of all the operations preserve causality and the op-groups form a total order, the order of execution of the op-groups is same at every participant and that order preserves causality. □

3.4 Generalizing the Algorithm

In order to generalize the algorithm to more than two classes of operations, assume a system of replicated state machines that support n different operations that can be invoked by client programs. Assume further that these operations can be subdivided into k disjoint sets $S_1, S_2, ..., S_k$, where any two operations within a set are commutative and any two operations from different sets are not commutative. Also, let different invocations of an operation in sets $S_1, S_2, ..., S_j$ $(j \leq k)$ be commutative. For such a system, define the *continuation properties* $\mathbf{C}_2$, $\mathbf{C}_3, ...,$ $\mathbf{C}_k$ as follows:

> **Property $\mathbf{C}_i$:** There is an unexecuted operation from every participant from the sets $S_i, S_{i+1}, ..., S_k$.

We observe that operations in set S_1 can immediately be executed if they are not in context of some unexecuted operations from sets $S_2, S_3, ..., S_k$, like the **lookup** operation in the previous section. In order to execute operations from set S_2, property $\mathbf{C}_2$ must be satisfied and the wave containing the S_2 operation must be complete. In other words, in order to execute operations in set S_2, a total ordering is required.

it is sufficient to show the following:

1. The same op-groups are formed at every process.
2. All operations in an op-group are executed consecutively, and the execution of all the operations in a noncommutative op-group follows the same total order at every process.
3. The order of op-group execution at every process is the same and this order preserves causality.

We formally prove each of these.

Theorem 1 *The same op-groups are formed at every process in the semantic dependent ordering protocol.*

Proof: We first observe that all the operations in an op-group are either commutative or noncommutative. First, assume that all the operations are noncommutative. Further assume that e_1 and e_2 are any two noncommutative operations. The op-group containing noncommutative operations is formed when the continuation property is satisfied and the wave containing commutative operations is complete. Since every process sees the same context graph, e_1 and e_2 are either in the same wave in every process' view or they are in different waves in every process' view. Thus, either e_1 and e_2 belong to the same op-group at every process or they belong to different op-group at every participant.

Assume e_1 and e_2 are commutative operations, but that at process P_1, e_1 and e_2 are in the same op-group, while at process P_2, e_1 and e_2 are in different op-groups. Also, without loss of generality, assume that e_1 is executed before e_2 at P_2 and that $u_1, u_2, ..., u_k$ are the noncommutative operations that are executed between the execution of e_1 and e_2. Since e_1 is executed before any of these noncommutative operations, e_1 either precedes or is at the same logical time as each of these noncommutative operations. Assume that e_2 is sent by process P_j. Noncommutative operations are executed when the continuation property is satisfied. In particular, in order to execute $u_1, ..., u_k$, there must be an unexecuted operation from participant P_j. This operation is either one of the noncommutative operations or e_2 itself. In both the cases, e_2 must follow at least one of the operations (say u_j) in $u_1, ..., u_k$. Since this relation holds at every process, e_2 must be executed after the execution of u_j at every process including P_1 and e_1 must be executed before u_j at every process including P_1. Thus, e_1 and e_2 cannot be in the same op-group at P_1. □

Theorem 2 *All the operations in an op-group are executed consecutively, and the execution of all the operations in a noncommutative op-group follows the same total order at every process.*

Continuation Property: There is at least one unexecuted operation from every process in the participant's view.

Essentially, satisfying the continuation property marks the completion of both the commutative op-group currently being executed and the succeeding noncommutative op-group *OG*. As a result, a sequence of op-group executions is potentially triggered when this condition becomes true. First, all the **update** operations in the current wave, which are the operations constituting *OG*, are sorted based on some deterministic criteria—e.g., based on the sender's identifier—and executed in that order. Then, the **lookup** operations that were deferred since they belong to the op-group following *OG* are executed. This action may, in turn, make the Continuation Property true for the next noncommutative op-group, and so on.

Main Loop:

If **lookup**
    If all predecessors have been executed
        Execute **lookup**;
    Else
        Defer execution and do **check_op-group**;
If **update**
    Defer execution and do **check_op-group**.

check_op-group:

While current wave is complete and continuation property satisfied
    Sort and execute all **update** operations in current wave;
    Execute all unexecuted **lookup** operations whose predecessors have been executed;
    Advance curent wave to earliest wave containing an unexecuted **update** operation.

Figure 4: Semantic Dependent Ordering Protocol

3.3 Correctness Argument

The algorithm essentially alternates execution between a sequence of commutative operations and a sorted wave of noncommutative operations. The sequence of commutative operations that are executed consecutively form a commutative op-group, while the sorted wave of noncommutative operations forms a noncommutative op-group. It is clear that each op-group contains either all the commutative operations or all noncommutative operations. To prove that the consistency criteria outlined in Section 3.1 is satisfied,

2. O is a maximal set of commutative operations, such that every operation in this set has the same set of op-group predecessors, where an op-group α is a predecessor of an operation e if $e \notin \alpha$, and some operation in α precedes e.

We refer to the first type of op-group as a *noncommutative op-group* and the second as a *commutative op-group*. As an example, for the set of operations (nodes) in Figure 3, the op-groups are $\{i_1\}$, $\{i_2, d_1\}$, $\{u_1\}$, $\{d_2\}$, and $\{ls_1, l_1, l_2, l_3, l_4\}$.

Now observe that for some op-group α that contains noncommutative operations and some other op-group β that contains commutative operations, one of the following two cases must be true: either the noncommutative operations in α precede zero or more of the operations in op-group β and are at the same logical time as the remaining operations in β, or the noncommutative operations in α are at the same logical time as one or more of the operations in β and follow the remaining operations in β. Thus, to maintain the directory object in a consistent state, the operations are processed as follows:

- All op-groups are processed in the same total order at every replica.
- The operations in a commutative op-group are executed in an arbitrary order.
- The operations in a noncommutative op-group are executed in the same total order.

For example, if we let α denote the op-group $\{ls_1, l_1, l_2, l_3, l_4\}$ in Figure 3, then the processors might sort the op-groups into the following total order: $i_1, d_1, \alpha, i_2, u_1, d_2$. The key idea is that while the processors must generate the same total order of op-groups, they may invoke the operations within each commutative op-group in an arbitrary order.

In the algorithm, operations in a commutative op-group are executed if all preceding op-groups have been executed, while operations in a noncommutative op-group are executed when its construction is complete and all its preceding op-groups have been executed. Without loss of generality, assume that there are only two operations: **lookup** and **update**. Figure 4 informally expresses the ordering algorithm; a more precise specification of the algorithm is given in [34]. Here, the current wave refers to the earliest wave in the context graph that contains an unexecuted **update** operation. The condition that all of a **lookup** operation's predecessors have been performed implies that it belongs to the commutative op-group currently being executed; in this situation, the **lookup** is executed immediately upon receipt. On the other hand, if this condition is not satisfied, the **lookup** operation is deferred. Completion of a noncommutative op-group is determined by the following property:

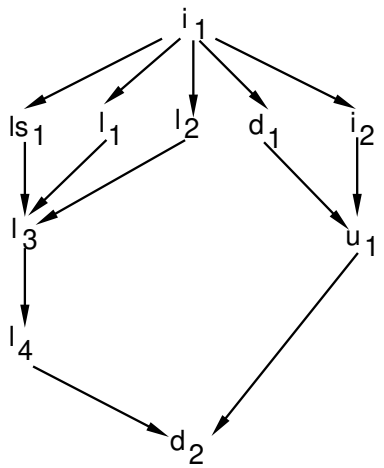

Figure 3: Example Partial Ordering of Operation Invocations

d_1 and i_2 must be executed in the same order by each directory manager and they must be totally ordered with respect to the group of commutative **list** and **lookup** operations. In other words, we assign a *precedence* to the operations and then use this precedence to break ties between operations that were invoked at the same logical time. For example, if **list** and **lookup** are at the same precedence level, and both are preferred to **delete**, which is in turn preferred to **insert**, then the set of operations ls_1, l_1, l_2, l_3, d_1, i_2 can be executed in any of the following orders:

$ls_1, l_1, l_2, l_3, d_1, i_2$
$ls_1, l_2, l_1, l_3, d_1, i_2$
$l_1, ls_1, l_2, l_3, d_1, i_2$
$l_1, l_2, ls_1, l_3, d_1, i_2$
$l_2, l_1, ls_1, l_3, d_1, i_2$
$l_2, ls_1, l_1, l_3, d_1, i_2$

In contrast, a solution based on a total order would have limited each manager process to just one total ordering, i.e., one of these six or some other that has d_1 and/or i_2 earlier in the ordering.

To understand how the algorithm works, consider the case where some of the operations are commutative. Intuitively, two or more commutative operations can be executed in any order—and in particular, in a different order by different manager processes—as long as there are no noncommutative operations "between" them. Formally, define an *op-group*, denoted O, to be a set of operations (nodes) in the context graph such that:

1. O contains all the noncommutative operations in a wave, or

3.2 Semantic Dependent Ordering Algorithm

One way to ensure replica consistency is to apply the various operations in a consistent total order. However, there are certain cases where such a strong ordering is not necessary. For example, the semantics of the application may allow a partial order for some operations, while requiring a total order for the rest. In such cases, a solution based on total order is overly restrictive since a pair of operations invoked at the same logical time—i.e., two messages (operations) sent by different clients—are coerced into a total order without regard for how the operations might or might not interfere with each other. A semantic dependent ordering avoids this unnecessary coercion by exploiting the semantics of the operations to provide an ordering that is weaker, yet still strong enough to maintain the correctness of the application.

Consul provides a semantic dependent ordering protocol in which operation commutativity is the semantic property exploited. The solution takes advantage of the partial ordering of messages preserved by Psync and knowledge about whether a given operation is commutative to allow two different managers to potentially execute the stream of operations in a different order. As as concrete example, consider the case of the replicated directory, where that operations **list** and **lookup** are commutative. Here, executing a collection of **list** and **lookup** operations in any order leaves the directory in the same state and returns the same set of results. Operations **insert**, **delete**, and **update** are not commutative, however, because applying them in different orders may leave the object in a different state. As a result, they must be executed using a total order.

In addition to this static relationship among the operations, there is also a dynamic relationship among the operations based on when they were invoked. Suppose Figure 3 represents the partial order of invocations of operations as given by a context graph, where the subscripts are used to distinguish between different invocations of the same operation. Here ls, l, d, i, and u represent **list**, **lookup**, **delete**, **insert**, and **update**, respectively. In this example, operations ls_1, l_1, l_2, d_1, and i_2 were multicast at the same logical time.

Our approach is to first order the operations based on the partial ordering—e.g., i_1 is executed before l_1 because it was invoked first—and then to take advantage of the commutativity of the operations to enhance concurrency. For example, because operations **list** and **lookup** are commutative and because ls_1, l_1, and l_2 were invoked at the same logical time, they can be executed in any order, and in fact, in a different order by each directory manager. Furthermore, because **delete** and **insert** are not commutative,

relationship, is directly provided by Psync. Semantic dependent ordering provides a message delivery order that exploits the semantics of the operations contained in the messages. In this section, we describe a semantic dependent ordering that is based on the execution commutativity of operations. Finally, the total ordering in Consul, called Total, is based on the partial ordering provided by Psync. The algorithm for this protocol is given in [38].

To make the discussion of our ordering protocols more concrete, we focus our attention on an application consisting of a replicated directory constructed using the state machine approach. The directory is a collection of name/value bindings on which the following set of operations is supported:

list(): return all the bindings in the directory.
lookup(name): return the value with the given name.
insert(name,value): insert a new name/value binding into the directory.
delete(name): remove the named binding from the directory.
update(name,value): update the named binding to have the given value.

The directory is replicated over multiple processors, with each copy being implemented by a directory manager process that maintains the data in volatile storage. Client programs, themselves distributed throughout the network, access the directory by invoking operations on one of the directory manager processes. This process then encapsulates the operation in a message and disseminates it to the other processes using Psync.

3.1 Correctness Criteria

Intuitively, the goal of the ordering protocols is to make the semantics of the operations applied to a set of replicas, in so far as the client can tell, exactly the same as if they were applied to a single replica. To do this, the order protocol must satisfy the following two conditions.

- The order in which the operations are applied to the various replicas leave all the replicas in an identical state.
- This ordering is consistent with the partial ordering of operations.

Note that this is only a general statement of correctness, since more precise conditions depend on the particular ordering protocol being used. Section 3.3 gives the precise statement for the semantic dependent ordering that follows.

Psync itself provides only minimal support for recovering from a processor failure. Specifically, Psync supports two operations that affect the local definition of ML. First, the local process can tell Psync to **maskout** a certain process. This operation removes the process from the local definition of ML and causes Psync to stop accepting messages from that process. Second, the local process can tell Psync to **maskin** a certain process. This has the effect of returning the process to the local definition of ML and accepting future messages sent by that process. Note that these operations are purely local; they do not affect the definition of ML at other processors.

In addition, a process can instruct Psync to **restart** a conversation. This operation is used to initiate recovery actions following a failure. Restarting a conversation has two effects: to inform other processes that the invoking process has restarted, and to initiate reconstruction of the local copy of the context graph. Psync accomplishes this by broadcasting a special restart message. When this message is received at a processor, the local instance of Psync performs two actions. First, it transmits its current set of leaf nodes to the participant that generated the restart message; this allows reconstruction of the lost context graph image using the standard lost message protocol. Second, it generates a local notification of the restart event; this is implemented as an out-of-band control message that is delivered to the local process. As an optimization, Psync also asynchronously writes the messages in the context graph to stable storage; this allows a given instance of Psync to recover the majority of its context graph from stable storage following a failure, with only messages not yet logged being retrieved from other processes over the network.

3 Ordering

The ordering protocols available in Consul allow messages from different processes in a group to be ordered in some consistent way at all receiving processes. The particular ordering used depends on the specific requirements of the application, but generally speaking, there are four possibilities: no order, partial order, semantic dependent order, and total order. Orderings at the beginning in this list are preferable since they provide faster message delivery and allow more concurrency, but they may not be strong enough to ensure the consistency of the replicas. Thus, the goal is to select the weakest ordering that still maintains the correctness of the application.

Consul provides all of these orderings. No ordering is the simplest kind of possible message ordering and no extra processing is required to provide such an order. Extra processing is required to provide the remaining three kinds of ordering. Partial order, in which message delivery is consistent with their causal

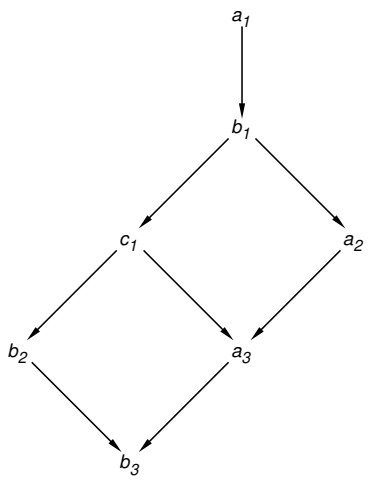

Figure 2: Example Context Graph

other are said to have been sent at the *same logical time*; for example, c_1 and a_2. Third, a message sent by a process is said to be *stable* if it precedes a message sent by all the other participants; for example, messages a_1, b_1, and c_1 are the only stable messages in Figure 2. Finally, a *wave* is a maximal set of messages sent at the same logical time, or equivalently, a set of messages such that the context relation does not hold between any pair. A wave is known to be *complete*—i.e., a process inspecting the context graph can be certain that no future messages will arrive that belong to the wave—as soon as a single message in the wave becomes stable. This is because for a message to be stable implies that all processes other than the sender have received it, which in turn implies that all future messages sent to the conversation must be in the context of the stable message; they cannot precede or be at the same logical time as the stable message.

Psync maintains a copy of a conversation's context graph at each processor on which a participant in the conversation resides. A distinct copy of the membership list ML is also maintained at each such processor. For simplicity, we assume only a single participating process is running on each of these processors. Each time a process sends a message, Psync propagates a copy of the message to each of these other processors. This propagated message contains the identifiers of all the messages upon which the new message *depends*; i.e., it identifies the nodes to which the new message is to be attached in the context graph. Psync recovers from network failures by recognizing when a new message is to be attached to a message that is not present in the local copy of the graph. In this case, Psync asks the processor that sent the new message to retransmit the missing message. That processor is guaranteed to have a copy of the missing message because it just sent a message in the context of it.

nodes labeled Network and Stable Store. These correspond to the asynchronous network and stable storage facility assumed by Consul. Similarly, User represents the application program that uses Consul. The rest of the protocols make up Consul proper: (Re)Start and Divider are involved in configuration management, Order ensures that messages are delivered to the application in a consistent order on all the processors, Membership is used to reach agreement about which processors have failed and recovered, Failure Detection monitors the activity of the participating processors and reports when it suspects one has failed, Recovery is used by a processor recovering from failure, and Psync provides a basic multicast service. A brief description of the (Re)Start and Divider protocols is given below. Psync has been described elsewhere in the literature [38], but since most of the other protocols depend on Psync, it is also briefly described below. The other protocols are described in the following three sections.

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 a proper message flow. Once these connections are established, the (Re)Start protocol remains quiescent, and is not involved in the normal functioning of the system thereafter. It becomes active again at the time of recovery and reestablishes these connections, which are lost when failures occur. The Divider protocol is a demultiplexing protocol that directs messages in the system to the appropriate protocols.

Psync supports a *conversation* abstraction through which a collection of processes exchange messages. The key idea of the abstraction is that it explicitly preserves the partial order of the exchanged messages. A conversation is explicitly opened by specifying a set of participating processes. This set is also called the *membership list*, ML. A message sent to the conversation is received by all the processes in ML. Fundamentally, each process sends a message in the *context* of those messages it has already sent or received. Informally, "in the context of" defines a relation among the messages exchanged through the conversation. This relation is represented in the form of a directed acyclic graph, called a *context graph*. Each participant has a *view* of the context graph that corresponds to those messages it has sent or received. Psync provides a set of operations for sending and receiving messages, as well as for inspecting the context graph. For example, Figure 2 depicts the context graph associated with a conversation that has three participants, denoted a, b and c, where $a_1, a_2, \ldots$ denotes the sequence of messages sent by process a, and so on.

Several important terms can be defined relative to the context graph. First, if there is a path in the context graph from one message to another, then the former message is said to *precede* the latter message; for example, a_1 precedes b_3 in Figure 2. Second, a pair of messages that are not in the context of each

2 Architecture

Consul assumes a distributed system in which multiple processors are connected by a communication network. There is no shared memory or common physical clock. Processes communicate by exchanging messages through the communication network, which is assumed to be 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. Processors in this system are assumed to suffer *fail-silent* semantics [39]; 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 [31].

A copy of Consul resides on each processor in the system, and provides an interface between the application program (replica) and the underlying network. These copies of Consul interact with each other to provide the relevant fault-tolerant services required for the state machine approach. In keeping with this approach, the system architecture is designed to handle only a relatively small number of replicas; thus, issues of scale are not addressed explicitly as they have been in other systems [5, 17].

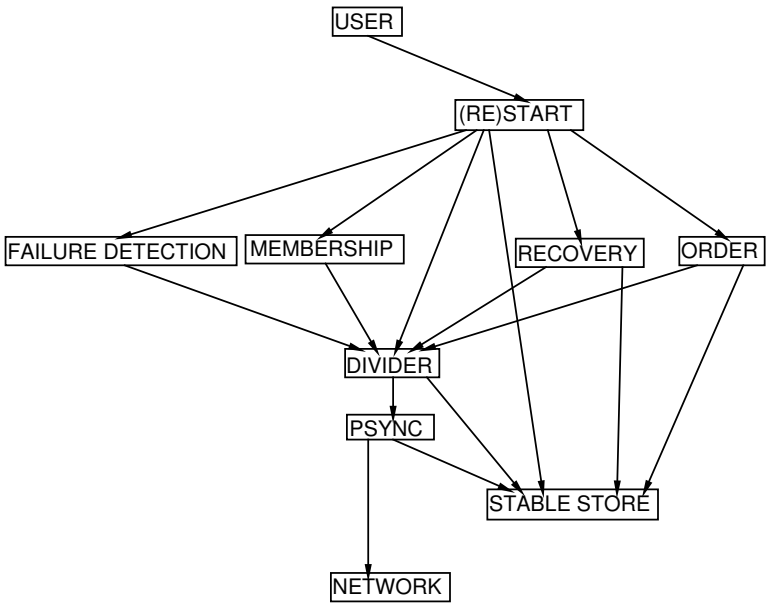

Figure 1: Structure of Consul

Consul is constructed from a collection of protocols, as illustrated in Figure 1, where each fault-tolerant service is implemented by one or more protocols. In this figure, an edge from protocol A to protocol B implies that A depends on B for correctness [11]. At the base of the protocol graph are

Resilience to failures is enhanced by replicating these state machines on separate processors in a distributed system. Among the key problems that must be solved to implement the state machine approach are maintaining replica consistency and reintegrating recovering replicas. Addressing these issues is simplified by the use of abstractions that we term *fault-tolerant services*. Examples of such services include *broadcast services* to deliver messages to a collection of processes reliably and in some consistent order, *membership services* to maintain a consistent system-wide view of which processes are functioning and which have failed, and *recovery services* to recover a failed process. Much work has been done on developing algorithms and implementations for these various services [3, 9, 10, 15, 25, 30].

Consul provides a unified collection of protocols that implement these fault-tolerant services. This collection of protocols, which forms a communication substrate upon which fault-tolerant systems can be built, provides support to manage communication, redundancy, failures, and recovery in a distributed system. Moreover, Consul is highly configurable in that a system builder is able to tailor an instance of Consul from these building block protocols. The Psync protocol, a group-oriented communication protocol that explicitly preserves the partial (or causal) order of messages, is at the heart of Consul [38]. One of the novel aspects of Consul is that the other building block protocols take advantage of the partial ordering of messages provided by Psync.

As a unified collection of protocols, Consul addresses two fundamental challenges in building a fault-tolerant distributed system. The first is understanding the interplay among the protocols that implement the various services. For example, the broadcast service and the membership service are not independent—the current set of running processes must be known in order to correctly order messages. The second challenge is accommodating the fact that different applications have different requirements. For example, some applications require that messages be totally ordered, while for others, a total order is overly restrictive. As another example, some applications assume at most one failure, and as a consequence, a system that tolerates multiple failures is overly restrictive. The bottom line is that a single monolithic system that locks in one set of design choices is likely to restrict the set of applications for which it is suitable.

This paper is organized as follows. Section 2 introduces Consul's underlying architecture, including a brief overview of Psync. The next three sections describe three of the building block protocols: Section 3 describes a protocol for ordering messages, Section 4 describes a protocol for agreeing on a set of running processes, and Section 5 describes a protocol for recovering from failure. Section 6 then reports on the performance of Consul and Section 7 discusses related work. Finally, Section 8 offers some conclusions.

Consul: A Communication Substrate for Fault-Tolerant Distributed Programs

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

This paper describes a communication substrate, called Consul, upon which fault-tolerant distributed systems can be built. Consul facilitates the development of such systems by providing various fault-tolerant services for constructing programs based on the replicated state machine approach. These services include a broadcast service, a membership service, and a recovery service. Consul is unique in two respects. First, its services are implemented using a collection of algorithms that exploit the partial ordering of messages exchanged in the system. Such algorithms are generally more efficient than those that depend on a total ordering of events. Second, its underlying architecture is configurable. That is, the application configures Consul from a collection of building block protocols, including protocols for ordering messages, detecting failure, agreeing on membership, and restoring the state of a failed process. The paper sketches Consul's architecture, presents the algorithms used by its protocols, and reports on the performance of an implementation in the *x*-Kernel.

Categories and Subject Descriptors: C.2.4 [**Computer Communication Networks**]: Distributed Systems—*distributed applications, distributed databases, network operating systems*; D.4.5 [**Operating Systems**]: Reliability—*checkpoint/restart, fault-tolerance*.

General Terms: Design, Performance, Reliability

Additional Key Words and Phrases: Partial Order, Configurable, Membership

1 Introduction

The design of a *fault-tolerant, distributed system*—a distributed system that can continue executing despite the failure of one or more of its processors—is an inherently complex task. To mitigate the impact of this complexity, various *programming paradigms* have been developed, each of which provides system organization techniques and abstractions for a given type of problem. These paradigms include the object/action model [19], the restartable action paradigm [31], and the replicated state machine approach [41].

This paper introduces a communication substrate, called Consul, that provides a platform for building fault-tolerant, distributed systems based on the replicated state machine approach. In this approach, a system is organized as a collection of servers implemented as state machines; each such machine accepts a sequence of deterministic commands and executes them atomically with respect to other commands.

[1]This work supported in part by NSF Grants CCR-8811423 and CCR-9003161, and ONR Grant N00014-91J-1015.