Consul: a communication substrate for fault-tolerant distributed programs

To cite this article: S Mishra et al 1993 Distrib. Syst. Engng. 1 87

View the article online for updates and enhancements.

Related content
- Supporting customized failure models for distributed software
  Matti A Hiltunen, Vijaykumar Immanuel and Richard D Schlichting
- Group communication in Amoeba and its applications
  M F Kaashoek, A S Tanenbaum and K Verstoep
- An application-level implementation of causal timestamps and causal ordering
  A Berry

Recent citations
- Dan Fleck et al
- Rüdiger Kapitza et al
- S. Mena et al
Consul: a communication substrate for fault-tolerant distributed programs

Shivakant Mishra†, Larry L Peterson and Richard D Schlichting
Department of Computer Science, The University of Arizona, Tucson, AZ 85721, USA

Received 14 July 1993

Abstract. Replicating important services on multiple processors in a distributed architecture is a common technique for constructing dependable computing systems. This paper describes a communication substrate, called Consul, that facilitates the development of such systems by providing a collection of fundamental abstractions for constructing fault-tolerant programs based on replicated processing. These abstractions include a multicast 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 (or causal) 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, thereby allowing a system to be structured according to the needs of the application. The paper sketches Consul’s architecture, presents the algorithms used by its protocols, and reports on the performance of an implementation using the x-kernel.

1. Introduction

Computers are increasingly being used in applications where dependable service must be provided despite failures in the underlying computing platform. While tolerating processor crashes and network problems is undoubtedly difficult, a number of well-understood techniques have been developed that can be used to make key computing services fault-tolerant. One such technique is to replicate processing activity on separate processors in a distributed system and then coordinate their execution so that they appear to the user of the service as a single logical entity. Thus, depending on the degree of replication and the type of faults to be tolerated, some number of processors can fail without the service being lost. This approach has been formalized as the replicated state machine approach [31].

This paper introduces Consul, a communication substrate that provides support for building fault-tolerant distributed programs based on the replicated state machine approach. This support takes the form of a collection of fundamental fault-tolerant services that simplify the dual problems of maintaining replica consistency and reintegrating recovering replicas following a failure. These services include a multicast service to deliver messages to a collection of processes reliably and in some consistent order, a membership service to maintain a consistent system-wide view of which processes are functioning and which have failed, and a recovery service to recover a failed process. These services are widely viewed as fundamental components of any fault-tolerant system based on replicated processing, leading to much work on developing various algorithms and implementations [24].

Consul provides these fault-tolerant services in the form of a unified collection of communication protocols. This collection, which forms a communication substrate upon which fault-tolerant programs can be built, provides support to manage communication, redundancy, failures, and recovery in a distributed system. Moreover, Consul is highly modular. Such modularization simplifies the development of new protocols, as well as increases the configurability of the resulting system [23]. The Psync protocol, a group-oriented atomic multicast protocol that explicitly preserves the partial (or causal) order of messages, is at the heart of Consul [26]. 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.

Consul has been implemented using the x-kernel, an operating system kernel designed for easy implementation and composition of communication protocols [14]. One version is based on version 3.1 of the x-kernel and runs stand-alone on Sun-3s, while another is based on version 3.2 and runs on Mach-based machines. Several applications involving replicating processing have been...
This paper makes two main contributions. The first is the presentation of new algorithms for implementing a variety of these fundamental fault-tolerant services. In particular, Consul provides novel realizations of the following:

- Consistent ordering of application requests to maintain replica consistency. Our approach exploits the semantics of the requests and the partial ordering provided by Psync.
- Membership. Our approach exploits the partial ordering of Psync and is especially efficient at handling multiple concurrent failures.
- Recovery. Our approach is based on the maintenance of a replicated context graph of messages by Psync.

The second contribution is the experimental evaluation of these algorithms as implemented in the x-kernel. This evaluation illustrates the way in which these algorithms can increase overall system concurrency and the minimal overhead they impose on normal operation. The net result is a system that provides the application designer with important new tools for constructing fault-tolerant distributed programs that use replicated processing.

This paper is organized as follows. Section 2 first describes the system model and gives an overview of Consul's architecture. Sections 3, 4 and 5 then describe three of the fundamental services provided by Consul: consistent message ordering, group membership, and process recovery. Section 6 reports on the performance of Consul, while section 7 compares our protocols with similar protocols proposed elsewhere. Finally, section 8 offers some conclusions.

2. Architecture

2.1. The replicated state machine approach

Consul supports construction of programs based on the replicated state machine approach [31]. In this approach, an application is structured as a state machine that maintains a collection of state variables. The state machine receives commands from other state machines, the environment, which prompt it to modify its state variables and potentially issue output commands. Execution of a command by the state machine is assumed to be deterministic and atomic with respect to other commands, so that a state machine's sequence of output commands is determined solely by its input sequence. A fault-tolerant version of the state machine is realized by running some number of replicas in parallel on different processors in a distributed system.

To ensure that the collection of replicas appears externally as a single logical entity, the dual problems of replica coordination and replica configuration must be solved. That is, all replicas must produce the same output in the same order, as well as deal with the failure and recovery of replicas in a uniform way. Other aspects of implementing the state machine approach, such as tolerating faulty clients and arbitrating among the replicas when producing output to the external world, are orthogonal problems not addressed here. Reference [31] describes techniques for handling these issues.

The state machine approach to replicated processing has been used over the years for a large number of applications (see [31] for examples) and is currently being used for the FAA's next generation air-traffic control system [10]. Because of its widespread applicability, the state machine approach is also supported by many fault-tolerant distributed systems, including Delta-4 [27], ISIS [6], MARS [16], and Transis [2].

2.2. Overview of Consul

As a hardware base, 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. Processes in this system fail by crashing; i.e., they fail silently without making any incorrect state transitions [28]. Finally, Consul assumes that stable storage is available to each processor, and that data written to stable storage survive processor crashes [19].

A copy of Consul resides on some number of processors in the system and provides an interface between the state machine 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. Client programs request service from the state machine by submitting a command to one of the replicas, usually the one closest to the machine on which the client is executing. This replica then uses Consul's multicast service to disseminate the command to the other replicas.

Consul is constructed from a collection of communication protocols, where each service is implemented by one or more protocols. Figure 1 gives the x-kernel protocol graph for Consul, where nodes represent protocols and edges represent a 'uses' relationship between the protocols. Specifically, an edge from protocol A to protocol B indicates that A opens B to send and receive messages on its behalf.

At the base of the protocol graph are nodes labeled Network and Stable Storage. These correspond to the asynchronous network and stable storage facility.

† The use of the term 'protocol' to refer to an implementation of a protocol in general and the nodes of this graph in particular is in keeping with standard x-kernel terminology [14].
assumed by Consul. Similarly, State Machine represents the application program that uses Consul. The rest of the protocols make up Consul proper: Psync provides a basic multicast service, Dispatcher, (Re)Start, and Divider are involved in configuration management, Membership is used to reach agreement about which processors have failed and recovered, FailureDetection 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 Order is a placeholder for different protocols that ensure messages are delivered to the application in a consistent order on all the processors (we discuss two specific examples—Total and SemOrder—in section 3). The configuration protocols aid the user in building a system according to the requirements of the application. These protocols establish connections among various protocols needed by an application for a proper message flow and for protocol operation invocations at the system start, and also re-establish these connections after a failure. These protocols are relatively minor, and so are ignored hereafter; for details, see [20]. Psync has been described elsewhere in the literature [26], but since most of the other protocols depend on Psync, it is briefly described below. The other protocols are described in the following three sections.

2.3. Psync

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 or causal 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. In the absence of any process failures, 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 process 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 participating processes, 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 $b_3$. Second, a pair of messages that are not in the context of each 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, $c_1, b_1,$ and $a_2$ are the only stable messages in figure 2. Finally, a wave is a set of messages sent at the same logical time, that is, the context relation does not hold between any pair of messages in a wave. A wave is known to be complete—i.e., a process inspecting the context 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 is done by sending either several point-to-point messages in a point-to-point communication network, or one multicast message in a broadcast network. 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. Mechanisms are provided for pruning the context graph or spooling messages off-line to stable storage under control of higher-level protocols.

Psync itself provides only minimal support for recovering from a processor failure. Specifically, Psync supports two operations that affect the local definition of ML. Note that these operations are purely local; they do not affect the definition of ML at other processors. First, the local process can tell Psync to mask out a certain process. This operation removes the process from the local definition of ML. It also causes Psync to stop accepting messages from that process. Second, the local process can tell Psync to mask in 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.

In addition, Psync supports a restart operation that is invoked by a process upon being restarted following a failure. Execution of this operation 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 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. Second, it transmits its current set of leaf nodes to the process that generated the restart message. The restarting process then uses a combination of messages spoiled to stable storage and the standard lost message protocol to reconstruct the local copy of the context graph. The restart message need not be broadcast atomically, although at least one process must receive it to ensure progress. This is achieved by broadcasting the message repeatedly at regular intervals until an acknowledgment in the form of a retransmit message is received.

Note that the reconstructed copy of the context graph may not be complete if messages were being pruned. In particular, the reconstructed graph will be the union of the graphs from all other functioning processors plus any messages spooled to local stable storage prior to the failure. Ensuring that messages needed for recovery of the application process are not pruned is the responsibility of higher-level protocols; see section 5.

3. Ordering

A key requirement of the state machine approach is that commands be received and executed in a consistent order at all replicas. The ordering protocols available in Consul provide this functionality by consistently ordering messages that have been multicast within the process group. The particular ordering used depends on the specific requirements of the application, but generally speaking, there are three possibilities: 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 these orderings. Partial order, in which message delivery is consistent with their causal relationship, is directly provided by Psync. Semantic dependent ordering provides a message delivery order that exploits the semantics of the state machine commands contained in the messages. In this section, we describe a semantic dependent ordering, called SemOrder, that is based on the execution commutativity of operations. Finally, the total ordering in Consul, called Total, delivers every message multicast in the group to all group members in the same order. This order is based on the partial ordering provided by Psync. The algorithm for this protocol is given in [26].

To make the discussion of our ordering protocols more concrete, we focus our attention on an application consisting of a replicated directory object constructed using the state machine approach. The directory object maintains a collection of name/value bindings and supports the following operations (i.e., commands) to modify the state or make queries.

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 object is replicated over multiple processors, with each replica managed by a process on one of those processors. Client programs, themselves distributed throughout the network, access the directory object by invoking operations on one of these manager processes. This process encapsulates the operation in a message, and disseminates it to the other manager
processes. We assume that operations that change the state of the object are implemented atomically; in other words, execution of the operation is guaranteed not to leave the entry being modified in an intermediate state despite failures. Techniques for implementing atomic operations can be found elsewhere [19,29].

To ensure the consistency of the replicas, some degree of consistent message ordering is clearly required. Total ordering is sufficient, but stronger than necessary in certain cases. For example, the semantics of the application may allow the partial order for some operations, while requiring a total order for other operations. A solution using total order is overly restrictive in such cases because a pair of operations invoked at the same 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 exploits the semantics of the operations to provide an ordering that is as flexible as possible, while maintaining the correctness of the application. Consul provides the protocol SemOrder that exploits the commutativity of operations used in certain applications. An operation is defined to be commutative if the execution of two or more consecutive invocations of that operation in any order gives the same result; an operation whose invocations are not commutative is called non-commutative.

The semantic dependent ordering presents an alternative solution to the problem of ordering operations on a replicated object. The solution takes advantage of both the partial ordering of messages preserved by Psync and the semantics of the operations, i.e., whether or not they are commutative. In other words, instead of enforcing a total order on the operations, violating this order when it seems appropriate, and using rollback to recover when the algorithm guesses wrong, our approach starts out with a weaker partial ordering and uses the knowledge about the commutativity of operations provided by the user to 'break ties'.

Consider now the replicated directory example, focusing on the three operations delete, insert and update. Among these operations, note that multiple invocations of delete are commutative assuming that deleting a non-existent entry is treated as a no-op. That is, executing a collection of such operations in any order leaves the object in the same state. Invocations of insert and update are not commutative, however, because applying them in different orders may leave the object in a different state.

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 d, i and u represent delete, insert and update, respectively. In this example, operations $d_1, d_2, d_3, i_2$ and $u_1$ were invoked at the same logical time.

![Figure 3. Example partial ordering of operation invocations.](image)

SemOrder first orders the operations based on the partial ordering—e.g., $i_1$ is executed before $d_1$ because it was invoked first—and then takes advantage of the commutativity of the operations to enhance concurrency. For example, because invocations of delete are commutative and because $d_1, d_2$ and $d_3$ were invoked at the same logical time, they can be executed in any order, and in fact, in a different order at each replica. Furthermore, because insert and update are not commutative, $i_2$ and $u_1$ must be executed in the same order by each manager process and they must be totally ordered with respect to the group of commutative delete 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 delete is preferred to insert, which is in turn preferred to update, then the set of operations $d_1, d_2, d_3, i_2, u_1$ can be executed in any of the following orders:

$$d_1, d_2, d_3, i_2, u_1$$
$$d_1, d_3, d_2, i_2, u_1$$
$$d_2, d_1, d_3, i_2, u_1$$
$$d_2, d_3, d_1, i_2, u_1$$
$$d_3, d_1, d_2, i_2, u_1$$
$$d_3, d_2, d_1, i_2, u_1$$

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 $i_2$ and/or $u_1$ earlier in the ordering.

To understand how SemOrder works, initially assume that none of the operations are commutative. In this case, the operations will have to be sorted into the same total order at each processor. This can be done by the Total protocol provided in Consul. Now consider the case where invocations of a particular operation are commutative. Intuitively, a sequence of such invocations can be executed in any order—and in particular, in a different order by different manager processes—as long as there are no non-commutative operations 'between'
them. Formally, define an op-group, denoted \( O \), to be a set of operations (nodes) in the context graph such that:

(i) \( O \) contains all the noncommutative operations in a wave, or

(ii) \( O \) 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 \( \alpha \) is a predecessor of an operation \( e \) if \( e \not\in \alpha \), and some operation in \( \alpha \) 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 \( \{i_1\}, \{i_2, u_1\}, \{i_3\}, \{u_2\} \) and \( \{d_1, d_2, d_3, d_4, d_5\} \).

Now observe that for some op-group \( \alpha \) that contains noncommutative operations and some other op-group \( \beta \) that contains commutative operations, one of the following two cases must be true: either the noncommutative operations in \( \alpha \) precede zero or more of the operations in op-group \( \beta \) and are at the same logical time as the remaining operations in \( \beta \), or the non-commutative operations in \( \alpha \) are at the same logical time as one or more of the operations in \( \beta \) and follow the remaining operations in \( \beta \). 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, and
- the operations in a non-commutative op-group are executed in the same total order.

For example, if we let \( \alpha \) denote the op-group \( \{d_1, d_2, d_3, d_4, d_5\} \) in figure 3, then the processors might sort the op-groups into the following total order: \( \{i_1\}, \alpha, \{i_2, u_1\}, \{i_3\}, \{u_2\} \). 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 SemOrder then, operations in a commutative op-group are executed if all the proceeding op-groups have been executed, and the operations in a non-commutative op-group are executed when its construction is complete and all its proceeding op-groups have been executed. This algorithm is expressed informally for the replicated directory example in figure 4, assuming for simplicity only one commutative operation (delete) and one noncommutative operation (insert). Here, a current wave refers to the earliest wave in the context graph that has an unexecuted insert. The condition that all the predecessors of a delete operation have been executed implies that the delete belongs to the commutative op-group currently being executed and so it is executed immediately on being received. If this condition is not satisfied, the execution of delete is deferred.

The completion of a non-commutative op-group is determined by the continuation property which is defined as follows:

**Continuation Property:** there is at least one unexecuted operation from every member process in the process' view.

Essentially, this condition marks the completion of both the currently executing commutative op-group and the following non-commutative op-group. This condition triggers a sequence of op-group executions. All the insert operations in the current wave (that constitute the following non-commutative op-group) are sorted using the same sort algorithm—e.g., based on the sender's id—and then executed in that order. This is followed by the execution of the delete operations that were deferred that belong to the op-group following this non-commutative op-group and so on. A more complete presentation of the algorithm and relevant correctness arguments can be found in [21, 20].

In the above, operations are grouped into only two classes: one in which invocations are commutative and another in which they are not. However, the approach can also be generalized to the case where there are multiple groups of operations, where invocations of operations within a group are commutative relative to one another, but non-commutative relative to invocations of operations in other groups. Specifically, assume the operations for a given replicated object can be subdivided into \( k \) disjoint sets \( S_1, S_2, \ldots, S_k \), where invocations of any two operations within a set are commutative and any two invocations of operations from different sets are not commutative. Also, let different invocations of a single operation in sets \( S_1, S_2, \ldots, S_j \) \((j \leq k)\) be commutative. For such a system, then, define the continuation properties \( C_2, C_3, \ldots, C_k \) as follows:

**Property \( C_i \):** there is an unexecuted operation from every participant from the sets \( S_1, S_{i+1}, \ldots, S_k \).

We observe that operations in set \( S_1 \) can immediately be executed if they are not in context of some unexecuted operations from sets \( S_2, S_3, \ldots, S_k \). Like the delete operation in the previous section. In order to execute operations from set \( S_2 \), property \( C_2 \) 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. Since a total ordering is required to execute operations in set \( S_2 \), we can execute all the operations from sets \( S_2, S_3, \ldots, S_k \) in the current wave as soon as property \( C_2 \) is satisfied and the current wave is complete. Thus, in cases where the objects have 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 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: FailureDetection, which handles
local detection of failure and recovery events, and Membership, which coordinates agreement among group members about the event. The FailureDetection protocol handles its task by monitoring 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 the process managing a given replica in some interval of time, while recovery is based on the asynchronous notification generated when the recovering process executes the Psync restart primitive. The FailureDetection protocol at a replica also sends a dummy message whenever the managing process does not send any message for some interval of time. This serves two purposes. First, it ensures that every message received by the process is acknowledged within some interval of time thus reducing the time for the message to become stable. Second, it reduces the possibility of an idle process being suspected as failed. Note, however, that the technique used in FailureDetection is independent of Membership, and so may employ any strategy to detect process status changes.

Membership is used to reach agreement among the currently executing processes that a failure or recovery event has occurred so that all replicas maintain a consistent view. This 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, Membership sits on top of Psync and coordinates the way in which processes modify their local membership list using the Psync primitives maskin and maskout.

### 4.1. Correctness criteria

As noted above in section 2, 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.
- The membership list is modified in such a way that properties such as stability of a message and completeness of a wave are *stable*, where a stable property is one that remains true after first becoming true.
- 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, these two conditions guarantee that context graphs and the 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.

It should be noted that in an asynchronous system, it is impossible for a process $p$ to distinguish between being partitioned from another process $q$ (i.e., being unable to communicate with $q$ within some constant time bound) and a failure of $q$. In our membership protocol, such a situation will result in multiple disjoint groups of processes, one for each partition. When this occurs, further execution is typically restricted to the group with a majority of processes. Although not implemented, a solution of this type is feasible in Consul given that processes know the size of the group of which they are a member.
4.2. Algorithm

For simplicity, first consider the case where at most one process (replica) 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 the initiation message sent by FailureDetection. 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. Membership 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. 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 process is decided to have failed, if $n - 1$ positive acknowledgements are received in response to the protocol initiation message. In case of recovery, the process is incorporated in 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. In the presence of such concurrent events, the protocol becomes much more complex. Perhaps the predominant reason for this is 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 itself 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 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 $S_I$ is said to follow another sf-group $S_J$ if the last message sent by any process in $S_I$ follows the last message sent by all processes in $S_J$. In this situation, the failure agreement for $S_I$ is typically performed after the failure agreement for $S_J$.

Notice, however, that it is also correct to perform the agreement for $S_I$ and $S_J$ 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 $S_J$ fail before they can participate in the failure agreement associated with $S_I$, or if a running process receives the protocol initiation message for a process in $S_J$ before receiving all messages associated with the protocol for $S_I$. 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:

(i) 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.

(ii) A recovering process is incorporated into the membership list at the same logical time as 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 follow $m_1$ 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 (p is down) message has been received. Similarly, a process $p$ is suspected up if it is not in the membership list and (p is up) message has been received.

Membership maintains two lists: SuspectDownList and SuspectUpList. SuspectDownList contains the list of (p is down) messages that have been received and SuspectUpList contains the list of (p is up) messages that have been received. As described above, 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.

- **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 marked 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 marked 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 task of Membership 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 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
Message | Actions of the Membership Protocol
--- | ---
(p is down) | If a message from \( p \) at the same logical time as \( (p \text{ is down}) \) has been received, then send \( (\text{Nack}, p \text{ is down}) \); otherwise send \( (\text{Ack}, p \text{ is down}) \), stop accepting messages from \( p \), insert \( (p \text{ is down}) \) in SuspectDownList, and adjust count.

\( (\text{Nack}, p \text{ is down}) \) | Remove \( (p \text{ is down}) \) message from SuspectDownList and start accepting messages from \( p \).

\( (\text{Ack}, p \text{ is down}) \) | If message \( (p \text{ is down}) \) is stable then decrement count.

\( (p \text{ is up}) \) | Send \( (\text{Ack}, p \text{ is up}) \), insert \( (p \text{ is up}) \) in the SuspectUpList, and increment count.

\( (\text{Ack}, p \text{ is up}) \) | If \( (p \text{ is up}) \) is stable, then incorporate \( p \) into the membership list, remove \( (p \text{ is up}) \) from SuspectUpList, and adjust count.

---

**Figure 6. Membership protocol.**

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 \( (p \text{ is down}) \) 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 \( (r \text{ is down}) \) is immediately followed by \( (\text{Nack}, r \text{ is down}) \).

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 \( (p \text{ is up}) \) or \( (p \text{ is down}) \) 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 left receives \( (q \text{ is down}) \) before the final \( (\text{Ack}, p \text{ is down}) \) message. This is possible because there is no path from the final \( (\text{Ack}, p \text{ is down}) \) to the \( (q \text{ is down}) \) message. Thus, both \( p \) and \( q \) are considered in one membership check period. In contrast, the process on the right receives all \( (\text{Ack}, p \text{ is down}) \) messages before receiving \( (q \text{ is down}) \). As a result, the rightmost 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. The correctness of this protocol is argued in [22, 20].

### 5. Recovery

The recovery service is concerned with bringing the state of a failed replica back into synchronization with the
The remainder of the replicas upon restart. The strategy currently implemented in Consul to realize this service is based on a combination of independent checkpointing and message logging technique. In this technique, replicas write checkpoints without attempting to coordinate with other replicas and messages are stored into a log on stable storage for later retrieval. When a failure occurs, the state of the replica is restored to the most recent checkpoint, with logged messages being used to make the final transformation to the current state. In the case of Consul, however, no explicit log is needed since messages are implicitly logged in the context graph when they are transmitted in what is, in essence, automatic sender-based logging [15]. This general technique is most applicable in situations where a failed replica restarts relatively quickly, either on its original processor after a reboot or some other functioning processor. For a replica that is down for an extended period of time, recovery based on state transfer from a functioning replica such as is done in ISIS [6] is more efficient. Such a strategy could be configured into Consul as an alternative recovery protocol.

In Consul, the recovery service is implemented primarily by the Recovery protocol, 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 copy 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 replica 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. Protocol checkpointing

In addition to the checkpoints written by the replica, the need to reconstruct the state of Consul itself dictates that certain protocols do their own checkpointing during normal operation. PSync, for example, needs to save enough information so that the local copy of the context graph can be rebuilt after a failure, so it periodically writes the local membership list ML and the unique conversation identifier to stable storage. As noted in section 2, PSync also spools parts of the context graph onto stable storage to minimize the message retransmission performed during the context graph rebuilding process.

In addition, Order checkpoints the view of the local process, a value that consists of a set of message identifiers. This checkpointing is done by Order only after all the messages in the view have been successfully received and their commands executed by the state machine replica; this guarantees that all preceding commands in the context graph have been applied as well. Failure Detection and Membership are stateless and hence, do not perform any checkpointing. The configuration protocols—(Re)Start in particular—save information to recreate the proper connections among various protocols used in Consul following a crash. The replica 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 restores the substrate to the checkpointed state, the second stage restores Consul and the state machine replica to the current state of the system, and the third stage initiates the membership protocol to incorporate the process back into the membership list. Since the context graph and its role as implicit message log make it perhaps the key component in Consul’s recovery strategy, we outline these stages by focusing on the context graph.

As shown in figure 9, the graph can be divided into four regions. Define wave(n-view) 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

<table>
<thead>
<tr>
<th>Stage</th>
<th>Actions at the recovering site</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stage 1</td>
<td><em>(Re)start: Collect checkpoints from all the protocols.</em>&lt;br&gt;<em>(Re)start: Restart each of the protocols with the required parameters.</em>&lt;br&gt;PSync: Reconstruct context graph from stable storage until the retrieved view.&lt;br&gt;PSync: If the retrieved view is not included in the context graph, send retransmit request for the missing messages.*</td>
</tr>
<tr>
<td>Stage 2</td>
<td>Recovery: Set every protocol in passive mode.&lt;br&gt;Recovery: Send restart message.&lt;br&gt;All protocols other than PSync function passively.</td>
</tr>
<tr>
<td>Stage 3</td>
<td>Recovery: If every alive process has acknowledged (p is up) message, set protocols to active mode.</td>
</tr>
</tbody>
</table>
Figure 9. Context graph at recovery.

$n$-view is any node in the newly restored view. Any commands 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 pt. Any commands 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 the graph consists of those nodes below wave(n-restart). Any commands here are new commands that $p$ will apply once recovery is complete.

It is clear from the above discussion that the commands 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 commands in the graph from below wave(n-view) up to and including wave(n-restart). This is done in the second stage 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. The reconstruction of the context graph is done by Psync in two steps. Recall that Psync spoils messages onto stable storage at regular intervals. In the first step, all messages in stable storage that precede some message in wave(n-view) or belong to wave(n-view) are retrieved and placed into the new copy of the graph. However, since the message spoiling and the checkpointing of the view by Order are done independently, the spoiled 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 missing messages from other functioning processes. Since all the commands 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 except Psync 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. Intuitively, 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 (Ack, p is up) 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 an (Ack, p is up) 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 Recovery knows which processes are functioning so that it can determine when all (Ack, p is up) have been received. To determine this, the recovering site processes messages associated with the failure handling protocols during the passive phase as it would 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 commands 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.

5.3. Recovery policies

While Consul provides the infrastructure to do recovery, a number of relevant policies are left to the application.
One example is the frequency with which to write checkpoints. Writing frequent checkpoints clearly minimizes the size of the second portion of the context graph and reduces the amount of duplicate work performed upon recovery, but at the expense of more overhead during normal operation. Another is deciding when old messages in the context graph can be pruned. The issue here is that an aggressive and uncoordinated strategy could thwart recovery by making it impossible to rebuild enough of the context graph to allow the failed replica to reconstruct its state. In particular, it must be guaranteed that the messages in the context graph back to the time of its last checkpoint—that is, \( \text{wave}(n\text{-}view) \) in figure 9—are available, either from being spooled to local stable storage or from the context graph copies on other processors. One simple strategy that realizes such a guarantee is for each process to send a multicast when writing a checkpoint, with only messages higher in the graph than the latest checkpoints of all processes being candidates for pruning.

6. Performance

Consul has been completely implemented. The code consists of approximately 10,000 lines of C code, of which 3500 is \( P \text{sync} \). Consul is implemented in the \( x \)-kernel, an operating system kernel designed explicitly for experimenting with communication protocols [14]. In our prototype implementation, stable storage is realized as an external global facility accessed over the communication network.

We have built two different applications using Consul: a variant of the replicated directory object described in section 3 and a distributed word game. Both 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 \( \text{SemOrder} \), while others use \( \text{Total} \). 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. A third application, a fault-tolerant version of Linda, is currently nearing completion. Consul is being used in this application to construct the language runtime system, and in particular, to implement \( \text{stable tuple spaces} \) by replicating the data using the state machine approach. Details can be found in [3].

This section reports on the performance of various protocols in Consul and the overheads they impose on the overall performance of the system. All the numbers reported here have been taken from the replicated directory object application running on a collection of Sun 3/75 workstations connected by a lightly loaded 10Mbs Ethernet. Various experiments were designed to measure the performance of \( P \text{sync} \) and the semantic dependent ordering protocol, as well as the overhead of the various failure handling protocols.

6.1. \( \text{Psync} \) timings

To measure the performance of \( P \text{sync} \), one byte messages were exchanged between a pair of user processes directly on top of \( P \text{sync} \). In this test, the resulting average round trip delay was measured as 2.9 ms. This number is derived by exchanging messages for 10,000 trips (20,000 total messages) and reporting the elapsed time for every 1000 round trips. Each of these measurements was then divided by 1000 to produce the average. Earlier experiments indicate multicast performance in the range of 4.5 ms for eight hosts [26].

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 application using \( \text{SemOrder} \) with the same application using \( \text{Total} \). 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 \( P \text{sync} \), \( \text{Divider} \), 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 \( \text{SemOrder} \), the average response time depends heavily on the overall mix of the commutative and the non-commutative 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 1000 operations. Each of these measurements was then divided by 1000 to produce the average response time. The execution time is measured for groups of 1000 operations to reduce the impact of measurement overhead on the results.

The results for the system configured for two replicas are shown in table 1. As expected, \( \text{SemOrder} \) improves the response time of the system as the percentage of commutative operations increase. The response time is 2.7 ms when all the operations applied are commutative, giving an improvement of about 25% over the use of a

<table>
<thead>
<tr>
<th>% of comm. operations</th>
<th>( \text{SemOrder} )</th>
<th>( \text{Total} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3.7</td>
<td>3.6</td>
</tr>
<tr>
<td>50</td>
<td>3.55</td>
<td>3.6</td>
</tr>
<tr>
<td>75</td>
<td>3.2</td>
<td>3.6</td>
</tr>
<tr>
<td>90</td>
<td>2.9</td>
<td>3.6</td>
</tr>
<tr>
<td>99</td>
<td>2.7</td>
<td>3.6</td>
</tr>
<tr>
<td>100</td>
<td>2.7</td>
<td>3.6</td>
</tr>
</tbody>
</table>
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 Membership, FailureDetection and Recovery. The code size for these protocols is about 1500 lines. However, these protocols are simplified to a large extent by the functionality of Psync. 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 non-commutative operations. The results are shown in table 3.

The overhead imposed by the failure handling protocols is about 0.6 ms per operation, or approximately 15%. This overhead is due to two factors. First, since the communication substrate includes more protocols, Divider has a larger set of protocols to which to demultiplex incoming messages. Second, the FailureDetection needs to receive every message exchanged in the system. Thus, Divider demultiplexes every message to both FailureDetection and Order.

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, 5000 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 the spooling to stable storage done by Psync and the checkpointing done by Order. At regular intervals, Psync spools all the messages received in the context graph since its last write onto stable storage. The message identifier, dependencies and contents are stored for each message. Order checkpoints the message identifiers of all the messages in the participant’s view. This checkpointing 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 and the semantic dependent ordering protocol is 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 s per 100 operations when the clients issue operations at 50 ops/s and the checkpointing interval is 1 s, while the overhead is 0.01 s per 100 operations when operations are issued at 10 ops/s. 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

<table>
<thead>
<tr>
<th>Ops/s</th>
<th>Checkpoint interval (s)</th>
<th>Time for 100 Ops (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>No checkpointing</td>
<td>10.0</td>
</tr>
<tr>
<td>10</td>
<td>5.0</td>
<td>10.0</td>
</tr>
<tr>
<td>10</td>
<td>2.0</td>
<td>10.0</td>
</tr>
<tr>
<td>10</td>
<td>1.0</td>
<td>10.01</td>
</tr>
<tr>
<td>20</td>
<td>No checkpointing</td>
<td>5.99</td>
</tr>
<tr>
<td>20</td>
<td>5.0</td>
<td>6.0</td>
</tr>
<tr>
<td>20</td>
<td>2.0</td>
<td>8.035</td>
</tr>
<tr>
<td>20</td>
<td>1.0</td>
<td>6.07</td>
</tr>
<tr>
<td>40</td>
<td>No checkpointing</td>
<td>4.0</td>
</tr>
<tr>
<td>40</td>
<td>5.0</td>
<td>4.05</td>
</tr>
<tr>
<td>40</td>
<td>2.0</td>
<td>4.15</td>
</tr>
<tr>
<td>40</td>
<td>1.0</td>
<td>4.25</td>
</tr>
<tr>
<td>50</td>
<td>No checkpointing</td>
<td>2.0</td>
</tr>
<tr>
<td>50</td>
<td>5.0</td>
<td>2.1</td>
</tr>
<tr>
<td>50</td>
<td>2.0</td>
<td>2.3</td>
</tr>
<tr>
<td>50</td>
<td>1.0</td>
<td>2.35</td>
</tr>
</tbody>
</table>
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 [5,7,25].

In the second category, the semantics of the application have been exploited to come up with a solution. In [11], semantic information has been used to implement a replicated directory, while in [12], 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 [13], semantic information is used to efficiently implement a multiversion time stamping 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 [18]. 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 a mixture of these different operations needs to be applied to an object. The approach proposed in [18] 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 [9,17]. All protocols of this type 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 [4,8,30] and the one proposed in this paper assume an asynchronous environment. The protocols proposed in [4,8,30] maintain a consistent view of the configuration, but have the property that 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 [30] 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 [16], AAS [10], Delta-4 [27] 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 US air traffic control 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.

Acknowledgments

We would like to thank the anonymous referees for their helpful comments. This work has been supported in part by NSF Grants CCR-8811423 and CCR-9003161, ONR Grant N00014-91J-1015, and DARPA Contract DABT63-91-0030.

References


