1 Introduction

This position paper recommends replicating the procedure call abstraction as a method for constructing highly available distributed programs. In order to make highly available systems much more widespread than they are today, we must make it easier for application developers to incorporate replication into their systems. Given that remote procedure call has proven to be a useful abstraction in building distributed programs [2], replicated procedure call seems to be an appropriate method for introducing high availability while hiding the complexities of replication. In this paper, we argue that the simplicity and familiarity of the procedure call mechanism make it an excellent model for introducing replication, and we discuss different choices one can make in designing a replicated procedure call system.

2 Motivation

Providing a general purpose mechanism for developers to build highly available applications is a necessary step before such applications will become commonplace. This is because replica coordination algorithms are subtle and complex. Without a general tool, developers must reimplement replica control algorithms for each application they build, and this is a time-consuming and error-prone task. Given that a general tool for replication is a good approach, we believe that the procedure call abstraction is appropriate for offering such a capability to developers. Remote procedure calls have attempted to preserve the semantics of local procedure calls while hiding the complexity due to distribution; similarly, replicated procedure calls can hide the details of replication while attempting to provide the same semantics as remote procedure calls. Many existing applications are structured using the client/server model with a procedure call interface. With a replicated procedure call, an application developer would not need to restructure the logic of the application in order to make it highly available. Since programmers are already familiar with the procedure call paradigm, a replicated procedure call provides a simple and natural extension that reduces the effort of building highly available services.

The procedure call abstraction is not appropriate for communication in all types of highly available systems, because it is not as flexible as a general message passing system. Nevertheless, the abstraction is often powerful enough for the developer’s needs while offering a simpler communication model. A common argument against the procedure call abstraction is that each call is synchronous, blocking the caller until the invoked procedure returns. Using a threads system can alleviate this problem. Another approach to solving this problem is the use of streams and promises to hide network latency [7].

Some have argued that the procedure call abstraction is not appropriate when multiple processes have a high degree of communication interdependence [4]. We suggest that communication interdependence is not the primary issue; rather, it matters more whether message recipients process messages in the same manner. That is, in systems in which one sender process sends the same request to multiple recipient processes, and the recipients process the request in the same manner (e.g., for fault tolerance), the effect is that of a replicated procedure call.

Another approach to supporting replicated servers is to use the programming model of process groups and broadcast protocols for communicating among the group members [1]. In order to implement replicated procedure calls in this model, the application programmer must do much of the work that would be taken care of by the stubs of a replicated procedure call system. Alternatively, one could present to the developer a general purpose replicated procedure call system built on top of the process group and broadcast
protocol model.

In summary, replicating the procedure call abstraction is appropriate as a vehicle for introducing high availability into distributed applications based on the client/server model.

CopyCat, the replication tool we are implementing at CRL, introduces replication at the procedure call level. CopyCat allows application developers to exploit the semantics of the service operations to relax the ordering constraints on the operations while maintaining consistency, thereby improving availability and response time [6, 5]. In his work on replicating procedure calls, Cooper provides a procedure call interface and guarantees consistency by ensuring all operations take place in the same order at all replicas [3]. In the rest of this paper, we investigate different choices one can make in designing a replicated procedure call system and discuss in particular how the choices were made in these two systems.

3 Operation semantics and behavior

Ideally, the introduction of replication should be totally transparent to the programmer. This requires that a replicated service appear to clients to provide the same behavior as an unreplicated service. One way to provide complete transparency is to force all operations to occur in the same order at all sites. This is the approach taken by Cooper's system. This choice introduces significant communication overhead and availability constraints (see below). In designing our system, we chose high availability and good response time as primary goals, thus influencing our decision to complicate the programming model slightly. In our model, a service may export three kinds of atomic operations: query operations that only read the service state, update operations that modify the service state but do not return results, and update-with-result operations. Distinguishing between updates and queries allows us to optimize our implementation of queries, so that they need not be performed at more than one replica. Furthermore, the distinction between updates with and without results is useful; updates without results need not delay the client until the operation is actually performed but may instead return as soon as the operation is stably captured.

Beyond this classification, we further distinguish between different types of updates depending on their semantics. Like others [1, 8, 9], we recognize that enforcing the same execution order at all replicas is not always necessary for maintaining consistency, and that better performance can be achieved by relaxing the ordering constraints. Therefore, we support three ordering types, which determines the relative order in which operations can be applied at each replica. Each update has one of the following ordering types: (1) causal - an operation for which the client may specify the updates that must precede it, (2) forced - an operation that, relative to other updates of this type, must be applied in the same order at all replicas, or (3) immediate - an operation that is performed at all replicas in the same order relative to all other operations.

To create a highly-available service, the service developer supplies a non-replicated implementation of the operations, and a classification of the operations in the service interface definition. The user of the highly-available service just invokes the operations and can ignore replication and distribution. In cases in which the three specific ordering types do not meet the developer's exact needs, it may be desirable to design an extensible system that provides a standard set of ordering types such as ours, but also allows application developers to specify their own ordering constraints as new situations arise. Supporting this extensibility is in contrast with the goals of simplicity and ease of use.

Operational behavior refers to the promises made to the client by the service about the progress of the invoked operation when the call returns to the client. Remote procedure calls usually offer the guarantee that, when the call returns, the operation has taken place no more than once, but the client may not be informed in case of failures. Cooper's replicated procedure call is based on troupes, which are groups of processes that comprise the replicated functional module; troupes can be both clients and servers. Cooper's system offers a client "exactly once execution at all troupe members [replicas]", with the client receiving the results of the invocation on each replica [3]. Our system ensures that update operations will take place at most once, even if the client submits the operation to more than one replica.

In CopyCat, an update without result is submitted to a single replica and returns as soon as the replica has registered the operation. There is no guarantee that the operation has yet been applied at any replica. The replica that registers the update is responsible for reliably propagating it to all other replicas, via gossip messages. For each update-with-result, we guarantee that when the call returns to the client, the operation has been applied at one or more replicas and will be applied when possible at all others. Queries are also processed by a single replica. Although a query re-

1These ordering types were formerly known as client-, service-, and global-ordered.
sult satisfies causality, submitting a query more than once might return different results. Since our system guarantees that both results satisfy the causality requirement, the client will still see a consistent view.

An RPC-based system may also provide support for nested calls. In our design for supporting nested calls, if each replica of a service \( S \) makes a call to another service \( S' \), all the calls from replicas in \( S \) appear identical to the replicas in \( S' \). Cooper’s system supports nested calls as well and has explicit support for replicated clients, which we did not perceive as important.

4 Availability and scalability

In our system, causal updates are available as long as the client can contact any replica. For forced updates, a majority of the replicas must be available, and for immediate updates, all the replicas must be available. A query will be delayed only if the available replicas do not have all the operations upon which the query depends. This situation becomes less likely if gossip messages are sent frequently.

Ensuring the correct ordering of operations at the replicas requires synchronization, either at the clients submitting the operations or at the replicas receiving the operations. We believe that as the size of the system increases, the number of servers grows much more slowly than the number of clients. In our system, synchronization occurs at the replicas; adding more servers increases the synchronization communication among the replicas, but adding more clients does not increase it. Since we expect our technique to work well in large systems, and since typically there will be large numbers of clients, this is an important consideration.

In Cooper’s system, peer members of a given troupe never communicate with each other; this requires synchronization at the client stubs. Thus, as the number of replicas in the service troupe grows, the computational demands for synchronization increase on the clients rather than on the servers. As the number of clients grows, the cost of synchronization increases as described below.

When supporting multiple client troupes, a single failure of any replica in a server troupe will halt the progress of the troupe commit protocol. Even without failures, the troupe commit protocol will deadlock whenever operations from different client troupes arrive in different orders at separate replicas within a server troupe, at which point deadlock detection is used to abort and retry the offending transaction. The likelihood of deadlock increases with the number of client troupes and with the number of replicas within a server troupe.

5 Conclusion

Widespread development of highly available systems will not occur until we reduce the difficulties of producing replicated systems. We advocate an approach that provides the developer with the high-level, natural abstraction of procedure calls. Message passing systems expose complexities that often complicate, rather than ease, the developer’s task. We encourage others to build systems that support the replicated procedure call abstraction, such as CopyCat, and to use them to incorporate high availability into distributed services.

References


dure Calls in Distributed Systems." Proc. SIGPLAN '88 Conf. on Programming Language Design and Implementation, 22-24 June 1988, Atlanta GA.
