Computing Replica Placement in Distributed Systems

A Position Paper for the Second Workshop on
the Management of Replicated Data

Daniel McCue
Mark Little
Computing Laboratory
University of Newcastle upon Tyne

Appeared in the Proceedings of the IEEE Second Workshop on Replicated Data, Monterey, November 1992, pp 58-61

Introduction

The actual gains in availability, reliability and performance which are achieved by object replication are a complex function of many factors including the number of replicas, the placement of those replicas, the nature of the transactions performed on the replicated object, the choice of replication protocols and the availability and performance characteristics of the machines and networks composing the system. For example, consider a naming or directory service offered in a local area network to record the locations of objects. To improve availability of this critical service, the directory object that provides the service might be replicated on several nodes in the network. However, if all of the nodes on which the replicas are placed receive power from the same power-point, the directory service has not increased its tolerance to power failures. Hence, availability is critically dependent on placement decisions and increases in replication level do not always result in higher availability.

We have been investigating the design of a replica management system (RMS) which allows a programmer to specify the quality of service required for individual replicated objects in terms of availability and performance. From the quality of service specification, information about the replication protocol to be used, and data about the characteristics of the underlying distributed system, the RMS computes an initial placement and replication level for the replicated object. As machines and communications systems are detected to have failed or recovered, the RMS can be re-invoked to compute an updated mapping of replicas which preserves the desired quality of service.

Preliminary simulation of our RMS has shown that its placement algorithm gives substantial improvements in the availability of a replicated service than simply placing replicas on nodes at random, as is done in most distributed systems. Our work has highlighted the importance of considering the failure characteristics of the nodes in the distributed system and failure dependencies among nodes when placing replicas, if the required availability for the replicated service is to be obtained (and maintained).

In [3] Cristian describes a similar service called the Availability Manager which attempts to maintain a level of availability by detecting the failure and recovery of replicas and adjusting the replication level accordingly. However, while the Availability Manager focuses on a mechanism for maintaining a level of replication, it does not directly address the issue of maintaining a level of availability: maintaining a constant level of replication does not ensure a constant level of availability. In fact, as we describe below, increasing the replication level may decrease availability.

The MARS system [8][5] is one of the most advanced, in terms of its consideration of placement decisions, in that it attempts to place objects at nodes which provide reliability characteristics consistent with the overall aims of the application. However, this analysis and placement is only performed at compile time and fixed for the duration of the execution of the application. While this approach is eminently sensible in the real-time, embedded systems applications at which MARS is targeted, we are considering a more general distributed system environment in which long-running application may require a dynamic, adaptive replication policy to ensure long-term availability.

Performance can also be affected by placement decisions and protocol choices. Consider the effects of placing replicas on unreliable nodes. The resulting unreliability of those replicas will generally require replica consistency protocols to work harder, increasing network message traffic and processing overheads. Not only will performance suffer as the replication protocol struggles to ensure that the replicas are consistent despite failures, but availability may actually decrease (despite the increased number of replicas) [7].

Coffman *et al* discuss the effects on system performance of varying the number of replicas in a distributed data base system [4]. They provide convincing evidence of the effect on performance of increasing the number of replicas, however, their work does not address availability effects or the effects of changing replication protocols.

Achieving increased availability by object replication

How is an appropriate replication level and placement determined for an object? In some systems, a programmer specifies a replication level for an object (e.g., 5 copies) leaving the run-time system to determine the placement of the five replicas in the network [2]. In other systems, a programmer may be able to specify the locations of the replicas [6][10]. These interfaces require the programmer to make a mapping from the desired characteristics of the (replicated) object, such as fixed level of availability, to a replication level and placement that will achieve those characteristics. Such approaches are inadequate for three reasons:

- A programmer has no possible basis for choosing *five* replicas, since availability does not vary proportionally with the number of replicas.
- The placement (as described) fails to take into account the reliability, performance or failure interdependences of the nodes on which the replicas are placed.
- The mapping is static, failing to compensate for on-going changes in network topology or load.

For a programmer attempting to improve the availability of an object by replicating it, the parameters which can be controlled are: the number and placement of replicas and the choice of replica consistency and recovery protocols [9]. The reliability of individual nodes and their failure interdependencies are parameters that cannot be controlled, but must be monitored so that their effects can be accounted for in replica placement policies.

Two gross measures of the reliability of a node are: the Mean Time To Failure (MTTF) and the Mean Time To Repair (MTTR). A node with a very low MTTF and a very high MTTR would be unsuitable for placing object replicas which will be active for a long time. To give values to these figures, which we call *reliability values* for a node, it is necessary to monitor the system components (nodes in this case) over time, and to continue to monitor them as applications run. A tolerance value can be computed for each statistic.

Failure independence for nodes in the network, often assumed in analytical work, simplifies calculations but rarely matches the reality of system configurations. Although a distributed system could develop a failure correlation matrix to attempt automatic detection of common failure patterns, one hopes that failures are not so common that a statistically significant result could be determined this way. Instead, we assume that failure dependencies which are of interest to a network administrator are entered "manually" into the policy process.

Computing object placement and replication level

The reliability values for individual nodes can be combined with other factors to compute an optimum placement of object replicas to maximize availability or performance. We expect that conflicts between performance and availability will generally be resolved by attempting a placement that will maximize performance within a certain availability limit. Thus, a user will generally specify only the availability factor, expecting the system to get maximum performance within that constraint. Hence we discuss first the necessary allocation for the desired level of availability, then the modifications to these choices to improve performance.

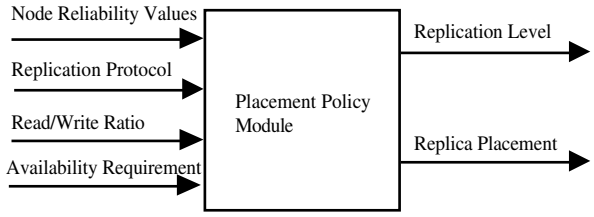


Figure 1 Input/Output of Policy Module

Figure 1 depicts the inputs and outputs of a *Placement Policy* module for managing availability, which would form the basis of the RMS. To make policy decisions for availability requires as inputs:

- The user's availability requirement for the object (e.g., should fail less than once a fortnight)
- The individual reliability values for all the nodes in the system (MTTF, MTTR)
- An estimate of the read/write ratio for operations accessing the object
- An indication of the type of replication protocol that will be used to maintain consistency between the replicas

The output of the placement heuristic is a replication level and placement (node list) which will achieve the desired level of availability. The availability of the nodes is computed from their MTTR and MTTF values and represents the probability that for the entire duration of a given operation (T_{end}-T_{start}) the node will be operational. We call this the *duration availability* for the node. The duration availability is a function of the probability that the node has never failed before T_{end} or that the most recent repair occurred before T_{start} and that the node has continued to function from T_{start} to T_{end}. The details of these calculations are given in [11]. The way in which individual nodes' duration probabilities are combined to calculate the overall probability of availability for the replicated object will be determined by the replication protocol. For example, using an Available Copies [1] replication protocol, which can tolerate k node failures with just k+1 replicas, the placement need only include a single node with a

duration probability greater than or equal to the required availability. However, in a majority voting protocol, a majority of the nodes must have a duration probability greater than or equal to the required availability. Furthermore, since some replication protocols behave differently for read operations than for write operations, these calculations must take into account the estimated read-write ratio for the object.

A tool for experimentation

We have constructed a simulation environment in which to experiment with various placement policies. Implemented in C++, this process-based, discrete-time simulator provides a programming environment similar to SIMULA's *Simulation* class and associated libraries. We experimented with several random number generators before settling on a shuffle of a multiplicative generator with a linear congruential generator which seems to provide a reasonably uniform stream of pseudo-random numbers. The co-routining behaviour of SIMULA, necessary for the process-based model, was implemented using the Sun Lightweight Process (LWP) library.

The simulated system consists of a set of nodes *N* on which a set of objects *O* is replicated according to our placement policy algorithms. Each object *O* is assigned a desired *availability factor* when it is created. A set of transactions *T* is executed, each accessing some subset of the objects for a fixed duration. The transaction arrival times and execution durations are each drawn from exponential distributions. Transactions that are interrupted due to a failure of a replicated object are restarted from the beginning. The measure of overall performance is average response time for completed transactions.

The nodes *N* fail and recover according to their MTTF and MTTR reliability values which are assigned when the nodes are "created" and never change during a single simulation run. In our initial experiments, node failures are assumed to be independent. The number of replicas created for each object is a function of the placement policy. We simulate two policies: A RANDOM policy in which the number of replicas is fixed by a programmer and the placement is selected at random, and a COMPUTED policy in which the number and placement are calculated as described above.

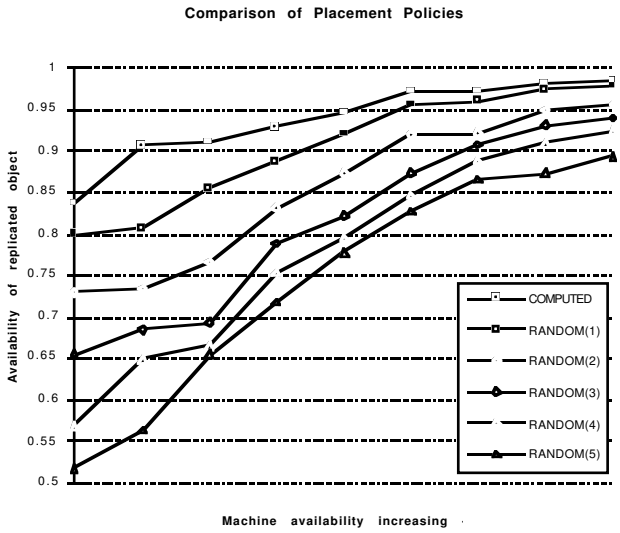
Our initial implementation of the COMPUTED placement policy assumes that the replication protocol is always the Available Copies protocol [1] and uses only the availability requirement for the object and the reliability values for the nodes to compute the number and location of replicas. Preliminary results from simulations using our primitive placement policy are promising, showing a consistent improvement in availability over that obtained by simply placing replicas randomly.

Preliminary results.

In our simulated environment we had ten machines in the network each with MTTF and MTTR values drawn at random. We estimate MTTR times to be approximately 1% of MTTF values. The desired availability from the replica group was 98%, i.e., for 98% of the operations performed on it by clients it should be available. Since we were simulating write availability on the object group, the COMPUTED policy attempts to keep the size of the group to a minimum, consistent with the desired availability.

Transactions were performed on the replica group using a read quorum of one replica and a write quorum of all replicas. If a failure of a replica occurs such that a transaction cannot complete then the transaction must be aborted and retried. Each simulation run performed over 10000 successfully completed transactions, and the results showed that the number of transactions attempted in the time differed greatly from the number of transactions which successfully completed, depending upon which placement strategy was used.

The figure below shows the results of the simulations. The six lines represent the results of the COMPUTED policy (top line) and the the RANDOM policy using 1,2,3,4 and 5 replicas (lower 5 lines). The lines converge on the right as the MTTF of the machines in the network was increased to the point where virtually all machines were up long enough to complete all transactions.



Effects of placement policies on write availability

Analysis

The COMPUTED policy consistently outperforms the RANDOM policy even when the number of replicas is the same. This is not surprising since the

COMPUTED policy searches for a set of machines that will satisfy the availability requirement. The COMPUTED policy fails to provide the desired level of availability however (which was 98% in all cases). This is due to the simplistic nature of the present algorithm and the fundamental limitation on write availability when the write quorum is “all replicas”. Current experiments involving quorum consensus protocols will provide a more realistic result.

Conclusions

The availability and performance of a replicated object is significantly affected by the placement and number of replicas and the choice of consistency protocol. Even crude calculations based on MTTF and MTTR of machines in a distributed system can improve the accuracy of placement decisions over random, i.e., programmer determined, placement decisions. More sophisticated techniques that take account of common modes of failure of nodes and the failure characteristics of other components such as the network itself could produce even better placement decisions.

With a tool for computing appropriate placements, a dynamic replica manager could be employed to take account of changing conditions in the distributed system such as changes in load or availability of machines. These dynamic placement decisions could ensure that characteristics such as availability and performance would be continually maintained over long periods of system operation.

The initial results from our simulations suggest promising directions for future research. In particular, we have identified several key parameters, consistency protocol, replication level and read/write ratio, which can be considered in placement decisions. Further work is needed in both analysis in estimating the availability of objects in this environment and experimentation with more sophisticated policies and more detailed models of distributed system behaviour.

Acknowledgements

The work reported here has been supported in part by grants from the UK Science and Engineering Research Council and ESPRIT project No. 2267 (Integrated Systems Architecture).

References

1. P. A. Bernstein, et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.
2. K. Birman, T. Joseph, F. Schmuck, "ISIS - A Distributed Programming User's Guide and Reference Manual", Department of Computing Science, Cornell University, Ithaca, NY, 1988.
3. F. Cristian, “Automatic Reconfiguration in the Presence of Failures”, Proceedings of the International Workshop on Configurable Distributed Systems, IEE, London, March 1992, pp. 4-17.
4. E. G. Coffman, E. Gelenbe, B. Plateau, "Optimization of the Number of Copies in a Distributed Data Base", IEEE Transactions on Software Engineering, Vol SE-7, No. 1, January 1981, pp. 78-84.
5. H. Kopetz, et al, "Tolerating Transient Faults in MARS", Proceedings of the 20th International Symposium on Fault-Tolerant Computing, 1990, pp. 466-473.
6. B. Liskov, "Distributed Programming in Argus", Communications of the ACM, March 1988.
7. M. C. Little, "Object Replication in a Distributed System", PhD Thesis, University of Newcastle upon Tyne, 1991.
8. M. Mulazzani, "Generation of Dependability Models for Design Specifications of Distributed Real-Time Systems", PhD Thesis, Technisch Naturwissenschaftliche Fakultät, Technische Universität Wien, Vienna, Austria, 1988.
9. J. D. Noe and A. Andressian, "Effectiveness of Replication in Distributed Computer Systems", Technical Report 86-06-05, Department of Computer Science, University of Washington, 1986.
10. B. M. Oki, "Viewstamped Replication for Highly Available Distributed Systems", PhD Thesis, MIT Laboratory for Computer Science, 1988.
11. K. S. Trivedi, "Probability and Statistics with Reliability, Queuing and Computer Science Applications", Prentice-Hall, Englewood Cliffs, NJ, 1982.