The Replica Management System: a Scheme for Flexible and Dynamic Replication

M. C. Little[†] and D. L. McCue[‡]

[†]*Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK*
[‡]*Xerox Corporation, Webster, New York 14580*

To Appear in the Proceedings of the Second Workshop on Configurabe Distributed Systems, Pittsburgh, March 1994.

Abstract

The actual gains achieved by replication are a complex function of the number of replicas, the placement of those replicas, the replication protocol, the nature of the transactions performed on the replicas, and the availability and performance characteristics of the machines and networks composing the system. This paper describes the design and implementation of the Replica Management System, which allows a programmer to specify the quality of service required for replica groups 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. As machines and communications systems are detected to have failed or recovered, or performance characteristics change, the RMS can be re–invoked to compute an updated mapping of replicas which preserves the desired quality of service. The result is a flexible, dynamic and dependable replication system.

Keywords: replication, reliability, configuration, system management.

1. Introduction

The placement of objects in a distributed system has a critical effect on the reliability and performance of applications using those objects. Objects with high reliability requirements should be placed on machines with high estimated reliability. Particularly important information may be replicated on several nodes of a network, employing some suitable additional protocol to ensure consistency [17]. The actual increase in availability that results from replication is a function of both the number of replicas and the placement of those replicas. For example, replicating a critical name–server object on three machines that receive power from the same wall socket will not increase tolerance to power failures.

Placing replicas on unreliable nodes will likely decrease performance (as the replica consistency protocol struggles to recover from failures) and may actually result in decreased availability. Co–locating objects will generally improve performance and, except for replicas of the same object, will generally increase reliability as well since it reduces the number of nodes that must remain operational for the application to complete successfully. Sometimes the choices involved in providing high availability conflict with high performance goals. If such conflicts are ignored, the application could suffer (e.g., the overhead imposed by having too many replicas could drastically reduce the performance of the application). Many of these factors are dynamic, changing as the system progresses and evolves. As such, a placement of an object which was correct when the application was started may be incorrect after it has been executing for some time.

We have been investigating the design of a *Replica Management System* (*RMS*) [20] which allows a programmer to specify a quality of service required for 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 links are detected to have failed or recovered, and user interactions with the object change, the RMS can be re–invoked to compute an updated mapping of replicas which preserves the desired quality of service.

This paper will describe the work we have conducted into identifying the various factors relevant to replica placement, and then how we have designed the RMS to take these into account.

1.1. Related work

In [8] Christian describes a 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 [13][21] 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 applications may require a dynamic, adaptive replication policy to ensure long–term availability and performance.

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) [17].

Coffman *et al* discuss the effects on system performance of varying the number of replicas in a distributed data base system [9]. 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.

Wolfson *et al* describe an algorithm in [30] for dynamically varying the degree of data replication depending upon the characteristics of a user's interactions. However, they do not consider either the implications of reliability of components on performance or the effects of using different replication protocols.

The rest of this paper is structured as follows: Section 2 briefly describes the types of replication protocol and the issues involved in replica placement. Section 3 then describes those issues which we have identified as being important to the placement of replicas, and therefore need to accommodate in the RMS. Section 4 describes the RMS and its associated infra–structure. Section 5 gives preliminary results and indications of future work.

2. Object replication

There are basically two types of replica consistency protocol: *active* and *passive* [17].

2.1. Active replication

Every functioning member of a replica group receives requests, performs processing, and sends replies. The active replication of an object requires the following two conditions to be met [16][17][26]:

- *Agreement*: All the non–faulty replicas of an object receive identical input messages;
- *Order*: All the non–faulty replicas process the messages in an identical order.

So, if all the non–faulty replicas of an object have identical initial states then identical output messages in an identical order will be produced by them (provided, of course, the computation performed by an object on a selected message is *deterministic*). This is the underlying principle of the *state machine* based approach to active replication [26]. Active replication is often the preferred choice for supporting high availability of real–time services where masking of replica failures with minimum time penalty is considered highly desirable.

2.2. Passive replication

Only one member of the replica group, the *coordinator* (*primary*), performs the processing and checkpoints its state to the rest of the secondary replicas (the *backups*) [2]. If the primary fails then a new primary is elected from amongst the backups.

One of the advantages of passive replication is that it can be implemented without recourse to complex, order preserving communication protocols. Because only a single member of the group processes any requests, the computation performed by the replicated objects need not be deterministic: the primary imposes its results upon the backups thus guaranteeing that all of the functioning members of the group possess the same state. However, one of the disadvantages of this protocol is that if the primary fails the time taken to elect a new primary can be a considerable overhead.

Since replica consistency protocols for object–oriented systems are relatively well understood [3][17][23], we shall not consider how such protocols function in the rest of this paper.

2.3. Using replication protocols

Assuming that a mechanism exists for generating replicas for an object and keeping them consistent, how could an application programmer make use of this facility to increase the availability of some critical object? In some systems (e.g., [5][7]), the programmer simply specifies a replication level (e.g., 5 copies), and the run–time system determines some random or pre–determined placement of the five replicas in the network. In other systems (e.g., [15][23]), the programmer may be able to specify the locations of the replicas. These approaches are inadequate for two 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 interdependencies 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.

The availability and performance of a replicated object is a complex function of many factors including (at least) the following:

- the number of replicas;
- the placement of replicas (i.e., the reliability of the nodes they are located on);
- the load average on the nodes the replicas are located on;
- the failure independence of the nodes in the network;
- the reliability of the system components;
- the consistency and recovery protocols used to maintain the replicas;
- any object interdependencies.

For a programmer attempting to improve the availability of an object by replicating it, the parameters which can typically be controlled are the number and placement of replicas and the choice of replication protocol [17].

The RMS which we have been developing provides a much better indication of the optimum number of replicas and their location within the distributed environment. In addition, because it can dynamically control the number and location of the replicas based upon the information given to it, it can take into account changes in network load, user interactions etc., as the applications run, typically giving better performance than would be possible from manual intervention.

3. Replica placement considerations

Before describing the RMS we shall first examine the factors which we have identified as relevant to replica placement, giving an indication of how we intend to take them into account when designing the RMS.

3.1. Component reliability

Throughout this paper, the term *component* will be used to refer to major hardware/software components of a distributed system, e.g., nodes (processors) and communication links (networks). The visible or useful reliability of a node is dependent on the reliability of related components in the system. For example, a workstation which fails once a year for a few seconds but is connected to the rest of the network by a communication link which loses messages every five minutes, will appear to be unreliable. Hence, it is necessary to consider the interactions between components to determine the overall reliability of a specific component. A *logical component* will be used to refer to a set of components which are so dependant upon each other that the failure of one means the failure of the rest with a very high probability.

3.2. Node reliability

Two gross measures of the reliability of a node are: the Mean Time To Failure (MTTF) and the Mean Time To Recovery (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 and which require very high reliability[1]. 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 to indicate its accuracy. In our system, nodes continually re–compute these values to keep them as up–to–date as possible. If a reliability value strays outside of the computed tolerances then a reconfiguration may be required.

It is often assumed in the literature that reliability is a definite quantity, consistently viewed by users, i.e., if user 1 determines that machine Z has a reliability of R_z then user 2 will also have the same reliability value for Z. This is a simplification which may not always be true. If Z in the above example fails roughly once an hour, and user 1 interacts with Z every 5 minutes, then it will observe Z's reliability to be $R_z{}^1$. However, if user 2 interacts with Z only once every two hours, it will determine Z's reliability to be $R_z{}^2$, where $R_z{}^2$ will typically be greater than $R_z{}^1$ because the probability of user 2 attempting to contact Z while it has failed is much less than the same probability for user 1.

We believe that by making the assumption that both user 1 and user 2 have the same view of the reliability of node Z (typically the most pessimistic view), this reduces the possible effectiveness of the distributed system. In the above example, machine Z may not be used as much as it could, because it is assumed to have a low reliability, with the result that the other machines in the network may become overloaded. Therefore, some means of specifying the *expected rate of interaction* (and its subsequent monitoring) is required. The expected rate of interaction can then be used as a weight to determine the *perceived reliability values* of the machines in the network.

1 There is an assumption in the use of MTTF and MTTR that past performance is a useful predictor of future performance – an assumption which may not be true for all systems.

3.3. Communication links

As with nodes, the reliability of communication links should likewise be monitored. However, although the statistics obtained can be expressed in a MTTF and MTTR manner, we believe it is more informative to express them as *probability of message loss* and *probability of network partitioning*. As with node reliability values, tolerances can also be assigned to these figures.

Message loss is typically not the result of problems with the communications media, occurring instead at the receiving node, for example because of message buffer overflow. However, being able to differentiate between causes of message loss is difficult and for simplicity we shall assume that is is a characteristic of the communications link. Further research is necessary to determine the exact effects of this assumption.

3.4. Common modes of failure

The reliability values described above give an indication of the reliability of only *individual* system components. They do not indicate dependencies between, for example, workstations. Failure independence for nodes in the network is often assumed in analytical work, because it simplifies calculation. However, it rarely matches the reality of system configurations. To express failure dependencies between nodes, we suggest two methods:

(i) the network administrator "manually" enters failure dependencies of interest into the policy process. We envision this taking the form of a directed graph, indicating the various nodes in the system and explicitly showing the dependencies between them, as shown in figure 1. We have identified several kinds of inter–node failure dependencies (common modes of failure) including: shared power source, common sub–network, common machine architecture/operating system, network disk dependencies (e.g., one node booting diskless from another or accessing critical software via a remote file system). These dependency arcs are labelled with a probability of common failure (i.e., 1.0 implies that a failure of the node at the head of the arc certainly causes a failure of the node at the tail of the arc). Many other kinds of failure dependencies can be defined. More research is needed to determine which kinds of dependency are most significant for any given network. For example, all of the interdependencies listed above are static properties of the network topology or the individual machines, but dynamic factors can also affect reliability: some studies indicate that the probability of failure of a node increases in direct proportion to the load. We are currently investigating other graphical display methods such as a false colour display or contour map overlayed on a representation of the distributed environment. Such a scheme may give a better indication of possible "hot spots" within the environment.

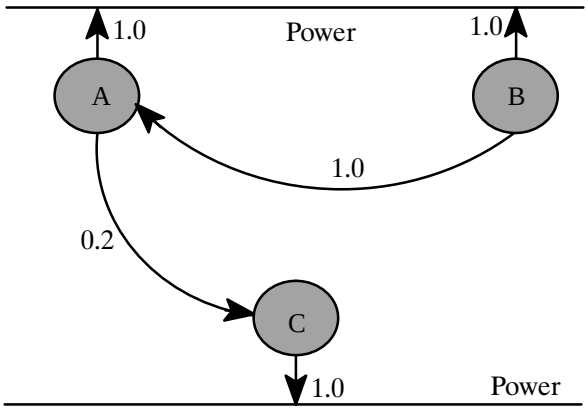

Figure 1: Graphical Representation of Node Interdependencies

(ii) the system attempts automatic detection of common failure patterns and develops a failure correlation matrix.

We believe that by combining these two methods it is possible to obtain a better indication of the dependencies within the distributed system. The system builds upon any dependency data input by the system administrator, continually refining it as new data are acquired. If no initial dependency data is given by a system administrator then obviously the RMS cannot know about any inter–node dependencies, and its initial placements may not be as accurate as they could be. However, as the system runs and data are obtained, the RMS develops a failure correlation matrix and re–evaluates object placements. To aid a system administrator, it is possible to obtain the failure correlation matrix from the RMS at any time. This can then be displayed and manipulated in a number of different ways before possibly being re–presented to the RMS.

3.5. Object interdependencies

Related to component interdependencies, and of perhaps more importance and greater likelihood are inter–object dependencies. Whenever one object invokes a method of another object then it can be said to be *dependant upon* that object. The degree of dependency (*degree of coupling*) of the objects is based upon many different factors including the frequency of method invocation and the properties of the object being invoked.

The dependencies between objects change more rapidly than machine interdependencies, and can vary from one execution of an application to another. This dynamic quality means that monitoring object interdependencies is a necessity as it can have an affect

on both the performance and availability of a given object. For example, as has already been mentioned, both performance and availability will typically be increased by co–locating objects which are dependant upon one another. If objects can be migrated from machine to machine (for example, by the RMS if it decides a reconfiguration is required) then the associated dependency data may indicate other objects that should also be migrated if performance and availability are to be maintained; in addition, the dependency information may indicate that such a migration is not possible (for example, the object to be migrated may be critically dependant upon another object which cannot be migrated). It may even be decided to migrate the *migrator* to the *migratee*.

3.6. Node performance values

The performance of a node is a complex function of at least the load on the node (i.e., the number of "active" processes executing on it) and the node's configuration (e.g., the speed of the processor and the amount of memory available to it). Load balancing techniques are well documented in the literature (e.g., [14][22]). However, since we are concerned with improved availability and performance of an object we require a more global picture of the distributed environment, rather than considering only the characteristics of a single node. For example, placing an object on an lightly loaded machine which is connected to the rest of the network by a low bandwidth communications link may, if the object requires distributed interactions, reduce the performance of the object and hence anything which makes use of it.

In addition, the object's dependencies as described in Section 3.5. also need to be taken into account. For example, if an object makes excessive use of other resources which have to reside on a (relatively) heavily load node, it may make more sense to place the object on that node: the overhead involved in placing the object on the overloaded node may be outweighed by the overhead incurred by remotely communicating with the required resources.

In general therefore, the resource utilisation patterns of the objects to be placed, and the resources provided by the nodes on which they can be placed, need to be taken into account. Obviously this may be difficult to do when an application is initially executed because its access patterns and resource utilisation mappings may evolve over time. However, a static mapping may well be better than no mapping at all. We are investigating ways of obtaining a dynamic mapping of object resource utilisation and corresponding resources available on machines in the distributed environment.

3.7. Communication link performance

The performance of a communication link can be expressed in terms of how fast it can deliver a given message (given its reliability value), the bandwidth it possesses etc. The requirements of different applications of the communications layer are expanding and changing rapidly to make use of the new communications media. For example, the new FDDI and ATM technologies offer advantages for real–time multi–media applications which did not exist before. Monitoring the performance of the various communications media available to a given distributed environment is essential if the performance is to be maintained [19]. At present we do not monitor the performance of the communications links in our system; we currently believe that, unlike machine performance, it is unlikely to be such a dynamic factor and as such can be statically allocated.

From the above discussion we can see that there is a great deal of information which is relevant to replicated object placement. Rather than continue to address each factor separately, in the rest of this paper we shall talk in terms of a component's *attribute values*.

We shall now describe the Replica Management System and how it obtains and uses the information we have mentioned to arrive at an optimum configuration for a replica group.

4. The Replica Management System

Figure 2 shows the replica management system for a single node, with the various components and their interactions indicated. In the following sections we shall examine each component separately and give an indication of how they interact.

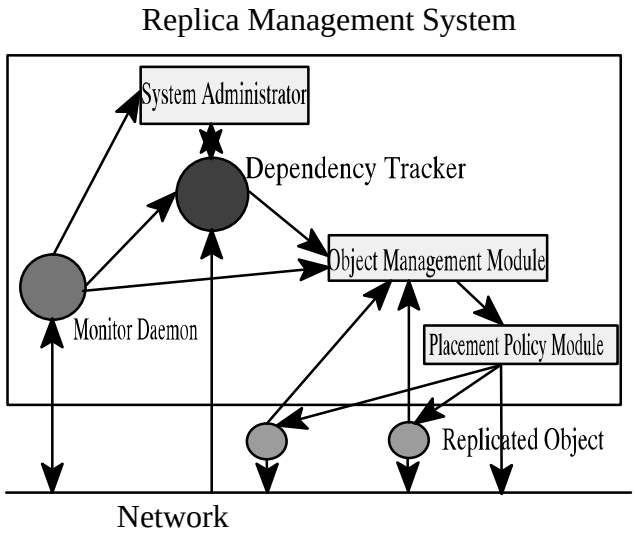

Figure 2: The Replica Management System

4.1. Acquiring and maintaining attribute values

For efficiency and fault–tolerance purposes the RMS should itself be available on multiple nodes. As such, we ensure that all of the relevant system attribute values are available on every machine in our distributed environment, regardless of whether an RMS is currently residing there. In this way, the time taken to create a new RMS is minimal.

Since nodes are physically remote from each other it is necessary to provide a means for one node to determine the attribute values of other nodes in the system. Similarly, when a new node is added to the system, it must discover the attributes for the other nodes in the system. This can be done in a variety of ways. We shall assume that a lazy distribution of attributes is used, e.g., once every 24 hours, each node in the system broadcasts to every other node its own attribute values (such as reliability values, average load, etc.), and their associated tolerances. Rather than insist that some reliable communications protocol is employed for this delivery, this communication can be done unreliably if required, since any node which does not receive a new value on time can ask for one after a timeout period, or can simply continue to use the (possibly) out–of–date values from a previous broadcast.

A new node which is to be connected to the system can simply contact any existing node to get a complete set of attribute values (since each node contains all attribute values), or could broadcast to all of the nodes and use the first (or most up–to–date) reply it receives. Since a new node should also broadcast its own attribute values, these can be piggybacked on to this initial request.

Recall that every attribute value has an associated tolerance. If a node determines that its attribute values have fallen out of the tolerances it originally specified, it can initiate an early broadcast to inform other nodes of the new values.

Note that the frequency of the attribute value updates can be altered to reflect the nature of the distributed system being considered. For example, in a very volatile network where nodes are continually being added and deleted or show characteristics that mean they do not stay within tolerances for long, it may be necessary to increase the frequency of these updates. If this dissemination of information is required to be delivered to all nodes in the system (for example, the network manager determines that each node's database of attribute values must be as up–to–date as possible) then a reliable communication protocol can be used, i.e., one which would ensure with a high degree of probability that all operational nodes will receive the information despite failures [6].

4.2. The Monitor Daemon

One of the key components of the RMS is the Monitor Daemon, which at regular intervals logs the current time and date, along with other attribute value information (e.g., current load average). This is similar to the Tatler described in [12]. Upon node recovery, the daemon examines the log and computes the MTTF and MTTR values. From other logging information, it is possible to determine whether the "failure" was due to a crash or a (possibly regular) "clean" shutdown. Being able to differentiate between the various causes of a node's unavailability is as important as being able to specify accurate MTTF and MTTR values.

In addition to being able to monitor a local machine's progress, a daemon can be instructed to monitor the progress of other machines in the system as well. This configuration information can be supplied either when the daemon is started or while it is executing. System utilities (e.g., certain disk partitions, mail application, etc.) can also be monitored. From this monitoring information, a reasonably accurate picture of the distributed environment can be built.

4.3. The Dependency Tracker

In addition to the Monitor Daemon described above, we have also designed and implemented a Dependency Tracker which takes as input the data from the various Monitor Daemons and outputs (in various graphical or text formats) information on the component dependencies which exist within the distributed environment. In addition to searching for dependencies based upon the "current" system data, it also checks for periodicities which may exist between failure dependencies by maintaining a complex history of data.

Such periodicities can be important because it may be possible to predict ahead of time that a given machine is going to be unavailable for a specific period of time, perhaps for maintenance, and move objects off it until it has been made available again.

The algorithm we use to determine possible dependencies is based upon *perceived availability* of nodes over a period of time. By collating all of the information from the various Monitor Daemons it is possible to build a *node availability* list for each node in the system. This is a list of all nodes which are agreed by all Monitor Daemons to have failed within the specified time period – it is necessary to use all of the daemons because it is possible that each daemon may possess a different view of the availability of other system components.

If a node *A* is unavailable *n*% of the time that a node *B* is unavailable then *A* is *dependant upon B* (i.e., *n* is a probability that the failure of *B* results in the failure of *A*). Note that this relation is not circular, i.e., in the above examine *B* may not be dependant upon *A*. After various experiments we decided on a value of *n* of 75%, which appears to give reasonable results (although this should be tuned on a per system basis and may need to be modified if the system configuration changes). We are investigating other algorithms for the dependency decision, possibly based upon the work described in [29].

4.4. The Placement Policy Module

The *Placement Policy Module* is responsible for computing the number and placement of the replicas. 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, typically a user will specify only the availability factor, expecting the system to get maximum performance within that constraint. However, we believe that some means of ranking availability and performance should be provided on a per application basis. This may mean that in some cases the user is more interested in having a particular application run faster at the expense of reliability.

There may also be a need for a means of ranking applications which share objects (e.g., different applications using the same objects may require different availability/performance constraints on those objects). At present the PPM will attempt a placement consistent with all requests; however, this may not always be possible and certain application requests may fail. If these are retried later, perhaps with a modified set of requirements, then the PPM may be able to achieve a placement consistent with every user's requirements.

4.5. The placement policy

Figure 3 depicts some of the inputs and outputs of the PPM for managing availability. To make this policy decision require:

- the user's required quality of service (e.g., "should fail less than once a fortnight");
- the individual reliability values for the components in the system (e.g., MTTF and MTTR);
- an estimate of the read/write ratio for operations accessing the object (the expected frequency of operations performed on the object can also be useful as described in Section 3.2.);
- an indication of the type of replication protocol that will be used to maintain consistency between the replicas.

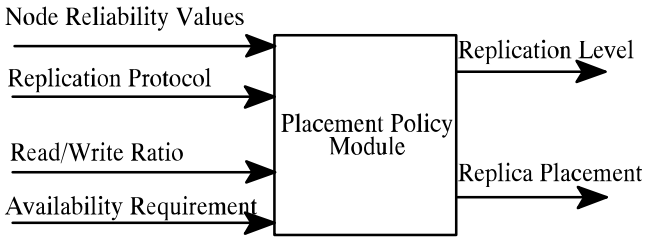

Figure 3: Placement Policy Module

The output of the *placement policy* 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 had 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 [28]. When individual node availabilities have been calculated, any node interdependencies must be accounted for to obtain a final availability value for each node, i.e., the *logical components* described in Section 3.1. form the basis for the placement model. We are currently investigating other placement policies such as those described in [24][25]. In addition, because the PPM could be modelled as a neural–net we are investigating the work described in [29] for a better algorithm which could learn more rapidly from its previous placement decisions.

The way in which the individual nodes' duration probabilities are combined to calculate the overall probability of availability for the replicated object will be determined by the replication protocol. (As was

mentioned in Section 3.2. the relative frequency of operations should also be taken into account, but for the purposes of this discussion we shall assume that the same duration availability values are observed by all users). For example, using an Available Copies [4] 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 [11], 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.

4.6. The Object Management Module

As applications are executing in the distributed environment, the attribute values for individual nodes may drift out of their tolerances far enough to warrant a re–distribution of objects (to maintain the desired levels of availability and performance). In this case some object replicas may be migrated from their current locations to other (perhaps more reliable, or faster) nodes, or new replicas may need to be created. If new nodes are added to the system it may become possible to downsize replica groups by moving replicas to these new nodes while still maintaining the required quality of service. The *Object Management Module* (*OMM*) is responsible for monitoring the system and determining whether or not a reconfiguration is required.

While research into distributed load–sharing has led to the conclusion that load changes may be too rapid and too frequent for practical adjustments [1][10], system reliability measures (e.g., MTTF and MTTR) tend to be fairly stable for relatively long periods of time. Hence, *availability–sharing* or adjustments to maintain a relatively constant level of availability should be possible. More replicas could be created, for example, if the number of read requests exceeded a certain level, and this would have the effect of reducing the load on individual replicas and improving the read response time. If the number of write requests increased then, assuming the availability requirement could still be met, it may be possible for the number of replicas to be reduced, thus improving the response time for write operations.

It should be possible for a user to change the values originally given to the PPM and to alter the constraints (perhaps to provide additional fault–tolerance). To support this “user–driven” migration will require some kind of user–interface to the management processes that control the placement of objects in the system. Although the RMS does not currently support this, we intend to add such an interface in the future.

4.7. Dynamic group changes

The only entities which can *accurately* determine the read/write ratio for an object are the objects themselves. Therefore, in our system the replicas are responsible for determining whether, for example, they are overloaded or not, i.e., the read/write ratio has exceeded the tolerance values originally set. As we shall see, such a decision need not be arrived at simultaneously, so no complex agreement protocol need be run between the replicas.

If the number of read operations outnumbers the number of write operations such that the current replica group cannot handle read requests in a timely manner, a replica may decide to inform the OMM, which will simply create new replicas to share the load, or move some of the existing replicas to faster machine. If write operations outnumber read operations then the OMM can reduce the size of the group and/or move replicas to faster machines. In both cases it is obviously necessary that the OMM ensure that this new group meets the same quality of service that the old group initially met.

If an appropriate protocol is in use, group membership can change dynamically, i.e., while the group is in use [16], or it can change when the group is quiescent.

4.8. Group change rates

Because no agreement protocol is executed between the various members of the replica group, it is possible that different replicas within the same group may come to different decisions about the size of the group required. This could lead to a situation in which newly created replicas are deleted by a replica in the same group. To overcome this synchronisation problem, we ensure that group membership changes are idempotent operations. This can be achieved by requiring each replica to specify both the current group size and the size it now requires when attempting to update the (global, replicated) group membership list.

However, this measure alone would not prevent oscillations in the size of the group in which two replicas with different views were alternating growing and shrinking the group. To control such oscillations, we introduce a form of hysterisis, imposing the restriction that once the group membership has changed it cannot change in the other direction for some time T. The value of T can be set on a per object/application basis.

Therefore, it is typically not necessary for replicas to synchronize with each other before requesting a group membership change. If one replica asks for the group to increase because it is overloaded but the other replicas are not, then this increased group size will eventually come to the attention of the other members (e.g., because their operations are running slower) and they will ask for the group size to be decreased. The selection of which replicas to eliminate when the group is downsizing will

again be a function of the availability/performance tradeoff and will invoke the same policy logic described previously.

5. Initial implementation and results

Before implementing any components of the RMS in a real distributed environment we decided to build a simulation of our system on which to experiment with various placement policy algorithms. The distributed systems simulation, described in [20] was written in the C++SIM simulation language [18].

The simulated system consists of a set of nodes *N* on which a set of objects *O* is replicated according to out placement policy algorithms. Each object *O* is assigned a desired *quality of service* 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 the overall performance is average response time for completed transactions (1.0 is the best performance possible), and availability is the number of successfully completed transactions.

The nodes *N* fail and recover according to their MTTF and MTTR reliability values which are assigned when the nodes are "created". These values are given a tolerance, and can change during a given simulation run – if they change by more than their allowed tolerance then, as described previously, this can trigger a reassignment of replicas. Node interdependencies can also be assigned either statically or dynamically as required.

In the simulated distributed environment there were 10 machines, each with MTTF and MTTR values which gave them availability characteristics between 98.9% and 99.2%.

Clients made use of objects for at least 10000 transactions. A client would start a transaction and invoke an operation on the (replica) group. If a replica failed during the operation the replication protocol determined whether sufficient replicas still remained operational for the client's operation to succeed (this obviously depends upon the "type" of the operation). If this was not the case then the client's transaction aborted to guarantee consistency.

With this distributed environment simulated we then constructed the RMS and experimented with various placement policy algorithms. We simulated two policies:

- A RANDOM policy in which the number of replicas is fixed by the programmer and the placement is selected at random – one replica is placed for each 20% required availability;
- A COMPUTED policy in which the number and placement are calculated by the RMS.

In order to compare the differences our RMS makes to the quality of service of a replicated object we first obtained results for replicated group interactions using three replication protocols based on Available Copies [4], Weighted Voting [11], and Passive Replication [2]. However, because of space limitations we shall show only those results for Available Copies. In the Available Copies replication protocol, users read from a *single* copy but must write to *all* copies.

5.1. Non–replicated interactions

Table 1 shows the availability and performance of a non–replicated object in our distributed environment. The single object was placed randomly on a machine with expected availability of 98.7%. The availability for both read and write operations is the same because it is the availability of the single object.

Type of Object	Availability	Performance
Single object	98.65%	1.0

Table 1: Availability and Performance of non–replicated object.

5.2. Availability of unshared replicated object

The quality of service we required for the replicated object was that it should be available 99.99% of the time and we required the best possible performance within this availability constraint. The read/write ratio was set to be 2:1, and there was only one client for the group.

Table 2 shows the results of the two placement policies. The RANDOM placement scheme assigned 5 replicas. Because the probability of all 5 replicas being unavailable for a given invocation is zero, the read availability for this group is 100%. The availability of the replica group for write–only operations is only 94.2% because the client must contact all of the replicas in the group, and therefore the probability that at least one will be unavailable is high. Because more replicas are used

for write operations the performance perceived by the client is subsequently worse.

Type of Operation	Availability	Performance
RANDOM Read−Only	100%	1.0
RANDOM Write−Only	94.2%	4.69
COMPUTED Read−Only	100%	1.0
COMPUTED Write−Only	98.2%	1.95

Table 2: Availability and Performance of Replicated Object.

The RMS decided that only 2 replicas were needed. As with the RANDOM placement, the read–only availability is 100%, and the performance is 1.0. Because less replicas are used for write operations, the performance is 1.95 and the availability is also improved to 98.2%.

5.3. Performance of shared replicated object

We next determined the performance of the same replica groups when multiple clients were involved (we decided to use 4 clients for this series of results). In this case, because our replicas are controlled by the familiar multiple–readers/single–writer locking policy, it is possible for some clients to find themselves unable to use the group because another client has locked it in a conflicting mode. The results are displayed in Table 3.

Type of Operation	Performance
RANDOM Read−Only	1.0
RANDOM Write−Only	4.8
COMPUTED Read−Only	1.0
COMPUTED Write−Only	3.8

Table 3: Performance of shared replicated object.

5.4. Dynamic reconfiguration

In the previous sections, the RMS statically allocated the number and placement of the replicas. As we have described, the real goal of our work was to obtain an RMS which could dynamically reconfigure the system to take into account changes in load etc.

In this experiment the read/write ratio is initially set to 10:1 and the required availability was set at 99.99%. Given this, the RMS computed that 4 replicas should be placed to give optimum performance and still achieve the best possible availability. However, the actual read/write ratio was changed during execution to become 2:1. As can be seen from table 4, if the alteration in read/write ratio is not accounted for, the overall performance of the group suffers as write operations must make use of all of the replicas in the group. When the group is dynamically reconfigured, the increase in the number of write operations results in the number and location of the replicas being altered with a benefit for performance.

Type of Placement	Performance
Statically allocated	2.3
Dynamically reconfigured	1.4

Table 4: Performance comparison for dynamic reconfiguration.

5.5. Current status

At present we are implementing the RMS on top of the Arjuna distributed system [27]. Because the RMS was fully implemented on the simulated environment we expect few problems in this translation. Both the Monitor Daemon and the Dependency Tracker have been implemented and tested on Arjuna and data from them has been fed back into the RMS running on the simulated network giving promising results. In addition we are considering extending the RMS as indicated in this paper and also to examine the effects of weak consistency replication protocols [12][17] on placement policy.

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 (both hardware and software) can produce even better placement decisions.

We have described the *Replica Management System*, a tool for computing appropriate placements, and dynamically reconfiguring a replica group to take into account the changing conditions in the distributed system, such as changes in load or availability of machines. These dynamic placement decisions ensure that characteristics such as availability and performance can be maintained over long periods of system operation.

Acknowledgments

This work has been supported in part by grants from the UK MOD and the Science and Engineering Research Council (Grant No. GR/H1078) and ESPRIT basic research project 6360 (BROADCAST).

References

1. G. S. Alijani, “Object Mobility in a Distributed Computer System”, PhD Thesis, Wayne State University, 1989.
2. P. A. Alsberg and J. D. Day, “A Principle for Resilient Sharing of Distributed Resources”, Proceedings of the Second International Conference on Software Engineering, 1976, pp. 562–570.
3. ANSA, “An Abstract Model for Groups”, ISA Project, APM/RC.259.01, June 1991.
4. P. A. Bernstein, V. Hadzilacos, and N. Goodman, “Concurrency Control and Recovery in Database Systems”, Addison–Wesley, 1987.
5. K. Birman *et a*l, “Implementing Fault–Tolerant Distributed Objects”, IEEE Transactions on Software Engineering, Vol. SE–11, No. 6, June 1985, pp. 502–508.
6. K. P. Birman and T. A. Joseph, “Reliable Communication in the Presence of Failures”, ACM Transactions on Computer Systems, Vol 5, No. 1, February 1987, pp. 47–76.
7. K. Birman, T. Joseph and F. Schmuck, “ISIS – A Distributed Programming User’s Guide and Reference Manual”, The ISIS Project, Department of Computer Science, Cornell University, Ithaca, NY, March 1988.
8. F. Cristian, “Automatic Reconfiguration in the Presence of Failures”, Proceedings of the IEE International Workshop on Configurable Distributed Systems, London, March 1992, pp. 4–17.
9. E. G. Coffman, E. Gelenbe, and 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.
10. D. L. Eager, E. D. Lazowska, J. Zahorjan, “The Limited Performance Benefits of Migrating Active Processes for Load Sharing”, ACM SIGMETRIC, 1988, pp. 63–72.
11. D. K. Gifford, “Weighted Voting for Replicated Data”, 7th Symposium on Operating System Principles, Pacific Grove, December 1979, pp. 150–161.
12. R. A. Golding, “Weak–Consistency Group Communication and Membership”, PhD Thesis, University of California, Santa Cruz, December 1992.
13. H. Kopetz *et al*, “Tolerating Transient Faults in MARS”, Proceedings of the 20th International Symposium on Fault–Tolerant Computing, 1990, pp. 466–473.
14. H. –C. Lin and C. S. Raghavendra, “A State–Aggregation Method for Analyzing Dynamic Load–Balancing Policies”, Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, May 1993, pp. 482–489.
15. B. Liskov, “Distributed Programming in Argus”, Communications of the CACM, Vol.31, No. 3, March 1988, pp. 300–312.
16. M. C. Little and S. K. Shrivastava, “Replicated K–Resilient Objects in Arjuna”, Proceedings of the 1st IEEE Workshop on the Management of Replicated Data, Houston, November 1990, pp. 53–58.
17. M. C. Little, “Object Replication in a Distributed System”, PhD Thesis, Department of Computing Science, Newcastle University, September 1991.
18. M. C. Little and D. L. McCue, “Construction and Use of a Simulation Package in C++”, Newcastle University, Department of Computing Science, Technical Report No. 437, July 1993.
19. D. D. E. Long, *et al*, “Providing Performance Guarantees in an FDDI Network”, Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, May 1993, pp. 328–336.
20. D. L. McCue and M. C. Little, “Computing Replica Placement in Distributed Systems”, Proceedings of the 2nd IEEE Workshop on the Management of Replicated Data, Monterey, November 1992, pp. 58–61.
21. 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.
22. H. Nishikawa and P. Steenkiste, “A General Architecture for Load Balancing in a Distributed–Memory Environment”, Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, May 1993, pp. 47–54.
23. B. M. Oki, “Viewstamped Replication For Highly Available Distributed Systems”, PhD Thesis, MIT Laboratory for Computer Science, August 1988.
24. K. Ramamritham, J. A. Stankovic, and W. Zhao, “Distributed Scheduling of Tasks with Deadlines and Resource Requirements”, IEEE Transactions on Computers, Vol. 38, No. 8, August 1989, pp. 1110–1123.

25. K. Ramamritham, J. A. Stankovic, and P. –F. Shiah, “Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems”, IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 2, April 1990, pp. 184–194.
26. F. B. Schneider, “Implementing Fault–Tolerant Services Using the State Machine Approach: A Tutorial”, ACM Computing Surveys, December 1990, pp. 299–319.
27. S. K. Shrivastava, G. N. Dixon, and G. D. Parrington, “An Overview of the Arjuna Distributed Programming System”, IEEE Software, January 1991, pp. 66–73.
28. K. S. Trivedi, “Probability and Statistics with Reliability, Queuing and Computer Science Applications”, Prentice–Hall, Englewood Cliffs, NJ, 1982.
29. C. –J. Wang, *et al*, “Intelligent Job Selection for Distributed Scheduling”, Proceedings of the 13th International Conference on Distributed Computing Systems, Pittsburgh, May 1993, pp. 517–524.
30. O. Wolfson and S. Jajodia, “An Algorithm for Dynamic Data Distribution”, Proceedings of the 2nd IEEE Workshop on the Management of Replicated Data, Monterey, November 1992, pp. 62–65.