Single System Abstractions for Clusters of Workstations

Bienvenido Vélez
Programming Systems Research Group
MIT Laboratory for Computer Science

Draft March 28, 2000

ABSTRACT

This paper is organized in two parts. The first part examines four approaches to provide the abstraction of a single system for a clusters of workstations: Berkley NOW [4], MOSIX [6], ParaStation [26] and Microsoft Cluster Service (MSCS) [24]. The criteria used to asses each system are transparency, availability and performance. The last section of this part proposes an alternative simpler design for MSCS that improves both performance and availability. The second part of the paper introduces the concept of *adaptive replication*, argues its particular relevance to clusters and its the compatibility with well-known replication algorithms and finally argues that a cluster-based replication service must support adaptive replication or face a steady (though low) decrease in availability as the number of nodes in the cluster increases.

Keywords: Workstation clusters, Networks of Workstations, Adaptive replication

1 INTRODUCTION

In this paper we define a *cluster of workstations* as a special type of distributed system in which a collection of interconnected whole-computers act together as a single computing system. Each computer or *node* in the cluster has local memory, disk and processing resources which are semi-autonomously managed by its local copy of some operating system. A number of additional simplifying assumptions become realistic in clusters of workstations. First, the cluster as a whole comprises a single administrative and security domain. Second, a higher level of node homogeneity (software and hardware) that in general distributed systems can be assumed. Third, cluster nodes can be assumed to be geographically closed. Finally cluster nodes can be assumed to be connected by fast network infrastructures.

This paper will not attempt to defend cluster of workstations as a superior architecture. For a fervent such defense the reader can consult [19]. Instead, the paper will examine different approaches to exploit the potential of clusters of workstations, for high performance and highly available computation. Some of the benefits of clusters, like scalability and availability, are directly inherited from its more general counterparts: distributed systems. But clusters have additional potential benefits derived from the stronger assumptions upon which they are based, namely:

Fast/reliable communication

Single administrative and security

Platform homogeneity (Hardware/OS)

Throughout the paper, I will keep in mind a number of demands that I believe the next generation of commodity distributed software will place on computing systems in general and clusters in particular. Some of this demands include:

Node Scalability Cluster software should graciously incorporate dynamic additions of nodes to the cluster. This increase in number of nodes should generate a corresponding in crease in system capacity, but should also increase (or at least maintain) availability.

365-24 Operation Cluster software should provide continuous operation.

Not all the systems that I will examine were originally designed with all of these demands in mind. Therefore, it will be unfair to criticize them for not achieving what their designers were not attempting to achieve. I will, however, attempt to unfold any potential limitations that these designs may present to their extension to satisfy the unexpected demands.

As a first step towards understanding and organizing the body of research under study I have defined a general scheme of three layers on which cluster-based systems can be characterized. This scheme is not intended to describe any real architecture of a cluster system. Instead, its usefulness lies on its ability to help me

Cluster Layer	System Image Implemented
Fine grain parallelism (FGP)	Massively parallel processor
Coarse grain parallelism (CGP)	Multi-programmed system
Network layer (NET)	Nodes with fast networks

Figure 1 Three layers of cluster-based software

understand the different problems being tackled by the various research efforts. The three layers, shown in Figure 1, are: the network layer (NET), the coarse-grain parallelism layer (CGP) and the fine-grain parallelism layer (FGP). Each can be viewed as exporting a system abstraction that can (but not necessarily) be used by the layers above it. The goal of the network layer is to export the abstraction of a system consisting of distributed nodes interconnected with very fast networks. The coarse-grain layer implements the abstraction of a single multi-programmed (e.g. Unix-like) system. Another name for this layer could be the network transparency layer. Many distributed systems in the past have attacked and solved different problems in implementing precisely this layer. Two of this systems are Sprite [18] and Amoeba [17].

The top-most layer (FGP) implements the abstraction of a massively-parallel processor (MPP). This layered approach has been directly followed by a number of real systems like PVM [11] and CYLK [23]. All these fine-grain parallel programming systems have been ported to some flavor of Unix, an implementation of the CGP layer.

My first use for this layered scheme will be to establish the focus of this paper. Together, the four systems under study comprise design efforts in all three layers above. A serious analysis of all the approaches to implement the different cluster software layers would require more time and space that I have been allocated. Although I will briefly explain the approach of each system to implement each relevant layer, I will limit my analysis to how each system implements the coarse-grain parallel layer.. In particular I will make emphasis on how each system exploits the potential of clusters of workstations to achieve high performance and highly available network transparency.

I have decided to focus the paper on the coarse-grain layer for a reason. I believe the demand for highly available multi-programmed systems is growing more rapidly than the demand for massively parallel processor abstractions. The advent of the internet as a global information repository is and important factor influencing this demand. The software infrastructure of the internet is all implemented at the coarse-grain level. Moreover, more and more content providers are becoming aware of the necessity of maintaining their sites operating continuously. In the internet there is no such thing as "business hours".

In the last section of this first part, I present an alternate design for MSCS that is simpler that the original design and has the potential to provide higher availability and performance.

The second part of the paper introduces *adaptive data replication* in the context of clusters of workstations. Data replication comprises the study of algorithms for maintaining multiple copies of data objects in order to improve both performance and availability in the presence of failures. In a cluster, failures can be expected to occur more frequently as the number of nodes is augmented to accommodate unexpected increases in load. A number of systems have been designed in the past to allow dynamic changes to the level of replication of individual objects. These changes can be triggered by either users or the system. However, to the best of my knowledge, no system in the past has been designed to maintain a desired *level of availability* as the number of nodes changes. I therefore introduce *adaptive replication* for clusters of workstations. Adaptive replication automates the process of determining the necessary changes in levels of replication necessary to maintain a desired level of availability in response to changes to the cluster composition.

2 Berkeley NOW

The Berkeley NOW (Networks of Workstations) project [3] focuses on exploiting the aggregate memory, disk and processing capacity of a cluster of workstations. The emphasis has been on achieving the best possible performance for both parallel and sequential applications that do not necessarily require very high availability. The project has been divided in several sub-projects each of which tackles fairly orthogonal problems. Three of these projects are Active Messages [25] , Server-less Network File Systems (xFS) [4] and Glunix [12]. The Active Messages project investigates techniques for providing fast interconnection infrastructures for clusters. The xFS project devised a fully distributed log-structured [20] network file system that stripes files [14] across cluster nodes in order to exploit their aggregate I/O bandwidth. GLUnix is a user-level software layer that provides transparent remote execution. Together, these three projects address the three layers of cluster software. I will focus this paper on GLUnix and xFS as

they present approaches to provide coarse-grain level abstractions for clusters of workstations.

2.1 xFS: A Server-less Distributed File System

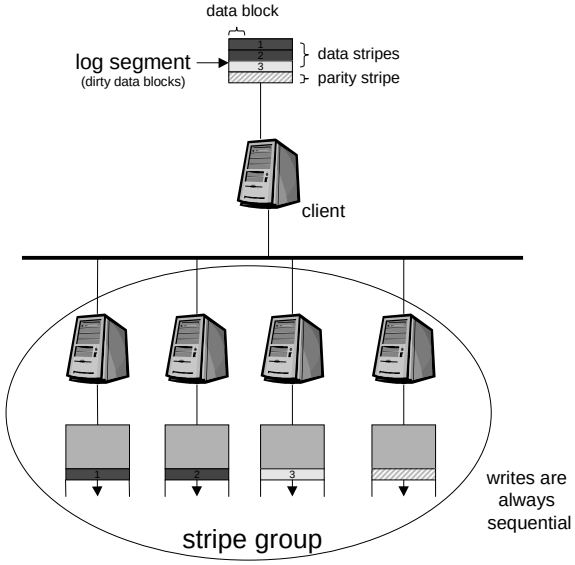

Figure 2 Distributed striped log segments in xFS

The main idea behind xFS is to implement a RAID storage system in software. As shown in Figure 2, xFS stores data in a log format. That is, all data is written sequentially at the end of the old data. The file system combines the advantages of log-based storage (performance, fast recovery) with those of stripping (availability and performance).

In xFS, any file (including directories) can be located anywhere in the cluster. In fact, a single file may be partitioned across different nodes. In Figure 2, a client stripes its set of dirty blocks, also called a *log segment*, into 3 data stripes. The client also computes a fourth redundant stripe containing parity bits that can be used to reconstruct any lost stripe from any other three stripes. The number of stripes is determined by the number of nodes in the corresponding *stripe group*. In the Figure, each server appends its new stripe to the end of its local log.

2.1.1 Virtues and limitations

Transparency

xFS scores high on transparency. It provides an abstraction of a single file system uniform across all nodes.

Availability

Redundant stripes increase the availability of the xFS system. They enable the file system to tolerate any single node failure. The degree of tolerance to network partitions in xFS is limited by the tolerance of the replication algorithm used to maintain the global data structures used to locate data. Unfortunately, the replication algorithm had not been published at the time of this writing.

xFS distributes both data and metadata (directories) across all nodes in the cluster. This has a significant impact on the complexity and, as is usually the case, the reliability of the system. In order to distribute metadata, xFS need to maintain several data structures necessary to locate information across all nodes. These data structures constitute global state that must be kept consistent and accessible to every node.

Log-structured file systems depends on ability of clients to accumulate large numbers of dirty blocks in order to assemble large enough log segments that generate large sequential writes to disk. This presents a reliability problem due to the potentially high risk of data loss as a result of a node failure. This appears to be an inherent limitation of the log-based approach. In [20] the designers of LFS assume that crashes are infrequent and users can afford to loose the last seconds or minutes of work. A pretty strong assumption indeed. In clusters, however, the frequency of failures increases linearly with the number of nodes.

Performance

The fully distributed organization of data in xFS has the potential to be transparently reorganized to balance the load and remove hot spots resulting from unpredictable usage patterns. The system can take advantage of the aggregate bandwidth of all disks and I/O channels. Implementing RAID in software avoids the need of RAID-based hardware to provide highly available storage.

2.2 GLUnix

GLUnix implements transparent remote execution with dynamic load balancing. In GlUnix supports the "home node" model of transparency originally proposed in Sprite [10]. In this model, users log-in into a single node, their home node, and from there execute programs that may end up running anywhere in the cluster. It is the job of the cluster software to guarantee that the job runs remotely as if it had run at the home node.

GLUnix provides the following services:

- Dynamic load balancing

- Channels I/O between the home and the execution node
- Allows GLUnix jobs to create children GLUnix jobs

A GLUnix job is not a normal Unix process, but rather a distributed collection if Unix processes. As a result, a user must wrap every Unix executable command with a GLUnix command (`glurun`) that encapsulates the knowledge about how to create a GLUnix job that can execute remotely in the cluster.

GLUnix also provides an API that processes can use to start GLUnix jobs.

2.2.1 Implementation

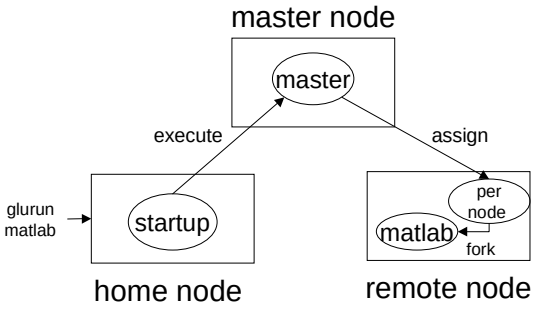

Figure 3 Remote execution in GLUNix

Figure 3 illustrates the process involved in creating a GLUnix job. The boxes represent nodes and the ovals represent processes. In the Figure, the GLUnix job is started by typing the command "glurun matlab" at the home node. This command creates a *startup* process inside the home node whose sole purpose is to serve a liaison between the user and the remote process doing the real work. The startup sends a request to a centralized *master* daemon process running at some well known node. The request includes the command "matlab" as well as environment data about the home node. The master determines which node appears less loaded and send a request to a per-node daemon running at the selected node. The per node daemon and creates a remote process that eventually executes the "matlab" command.

Local node load information is periodically computed by the per-node daemon processes and propagated to the master.

GLUnix runs as a user-level system on top of Unix. It consists of the following components:

Per-cluster master daemon A centralized process that coordinates communication among all the per-node daemons and GLUnix processes. It maintains a central repository of load information as well as a central mapping from global process identifiers (pids) to Unix pids local to nodes. Detects failures of nodes via timeouts and coordinates garbage collections of processes related to those that were running on the failed node.

Per-node daemon This per-node daemon process gathers load information local to the node and communicates it to the master daemon. The per-node daemon also carries out local actions (e.g. jobs stop and restart) on behalf of the master daemon.

API Library This library implements the API used to create cluster-aware applications. Includes facilities for spawning processes, process communication thru signals and barrier synchronization. The API implements a global name space for process ids.

Commands GLUnix provides several shell commands that allow users to execute and manage GLUnix jobs.

2.2.2 Virtues and limitations

Transparency

GLUnix provides home node transparency. What this means is that to a user or program spawning a job the cluster appears as the single multiprogrammed system where the job is created. This type of transparency is not ideal because users and programs must still be aware of the fact that the cluster is a collection of systems. Home node transparency puts the burden of maintaining a uniform image of the system across all nodes on higher level subsystems and the system administrator. A slightly higher level of transparency could be achieved if home nodes were determined on a per-user basis independently of where the users happens to be logged-in. This approach provides a consistent abstraction of the cluster to each user which can result in more predictable behavior from the cluster. Users could be assigned home nodes according to different policies. Of cource, the problem with this approach is that the cluster may look different to different users, but such was the case when home nodes correponded to process creation points.

Another limitation to achieve transparency from the point of view of processes in GLUnix is caused by the lack of kernel support. After a process gets started at a remote node, all the system calls that it generates are

handled by the remote kernel. Such system calls may expose information about the environment at the remote node that is inconsistent with the environment at the home node. MOSIX fixes this problem by redirecting potentially location dependent system calls to the kernel at the home node. This mechanism is described in more detail in section 3

Availability

GLUnix achieves some level of fault tolerance by automatically detecting, via timeouts, failures of the per-node daemon or startup processes. The master is totally responsible for detecting such failures and keeping information about which processes are alive. Upon detecting a failure, the master coordinates the process by which any remnants of GLUnix jobs running on the failed node are garbage collected. GLUnix cannot tolerate failures of the master daemon and it provides no facility for automatically restarting failed jobs. Thus, although failures are fairly isolated and the system as a whole tolerates failures, the individual *services* provided by nodes are not fault-tolerant.

Although the paper does not describe it, seems to me like GLUnix could easily be extended support dynamic addition of new or recovered nodes to the cluster. This will require a mechanism such as heartbeats (see MSCS below) to be implemented by each per-node daemon. Tolerating failures of the master daemon would require more sophisticated mechanisms. A new master node would have to be elected and the master daemon state recovered.

As a user-level solution, GLUnix is much less dependent on the underlying operating system and is compatible with old executables. Although GLUnix uses load information to decide where to run jobs, it offers no mechanism for preemptively migrating a running job to another node in response to changes in system load distribution. Therefore it can only hope to achieve limited load balancing.

Performance

GLUnix's centralized approach to keeping the status of GLUnix processes is simple and avoids the complexity of replicated data management. It, however, represents a single point of failure in the system as well as a potential performance bottleneck for clusters with many nodes. The authors present some evidence demonstrating that the centralized master daemon is far from becoming a performance bottleneck in a configuration of 100 nodes. This result may indicate that a centralized approach combined with a fault-tolerant mechanism like process pairs [7] to replicate the master daemon may offer a suitable solution for environments where communication with the master is not frequent. In GLUnix, communication flows to the master from startup processes (to forward signals and start jobs) and per-node daemons (to report load information).

3 The MOSIX Distributed OS

The main goal of the MOSIX [6] project is to implement *preemptive load balancing* in clusters of workstations. Preemptive load balancing attempts to maintain every node in the cluster equally busy by reassigning processes from heavily loaded nodes to less loaded nodes at any time during their execution. Every process in MOSIX starts to run at some node, called the *home node*, determined by the user or parent process that started the job. However, process may migrate through several nodes several times during its lifetime.

The designers of MOSIX selected a preemptive load balancing approach because the foresaw large and hard to predict variations in resource usage by processes. Other simpler approaches to load balancing include *static load balancing* and *dynamic load balancing*. In both cases, processes run to termination once they are assigned to a node. The difference between both approaches is that dynamic load balancing continuously collects load information from all nodes. Assignments of processes to nodes is based on the latest information collected. Static load balancing works well when the resource demands of each process are known or can be accurately predicted ahead of time. Dynamic load balancing predicts future resource demands based on current resource demands and uses this information to adapt its process assignment decisions. In neither case is it possible to relieve a node overloaded with poorly assigned processes by migrating work to other nodes. If processes are relatively short lived, the complexity of a process migration mechanism necessary for preemptive load balancing may not be warranted.

MOSIX provides the illusion that the home node is a powerful single multiprogrammed workstation. Therefore, MOSIX implements the CGP layer of the layered cluster model (Figure 1). MOSIX is a complete operating system providing the same user interface provided by other versions of Unix. The main difference is that users do not need to specify special commands to run processes remotely. Instead, they start their processes the same way they would to run them locally. Based on information about the load of each node, MOSIX

Process migration in Mosix is completely transparent and universal. *Any* process can be migrated at any time and, after migration is completed, will continue to execute as if it was running at the node that it was originally started on. This behavior is guaranteed for both interactive and batch jobs. Any process can run any-

where in the cluster. This high level of transparency comes at a price in both performance and portability which will be discussed in the "Virtues and Limitations" section below.

3.1 Implementation

This section describes the load balancing algorithms used by MOSIX. In MOSIX, process migration can be triggered by the one of following events:

1 a process explicitly requests to be migrated

2 a node detects that migrating a process may improve load balance

3 a node detects excessive remote I/O by a process

4 a node detects excessive forking by a process

5 a node shutdowns

I have been unable to find a description of the specific algorithms used to determine events (3) and (4) above. Events (1) and (5) are not triggered automatically by MOSIX itself. Therefore, I will delimit by discussion to how MOSIX migrates processes in response to load imbalances. My description is based on [5].

The load information necessary to determine how to reassign processes to processors is gathered independently by each MOSIX node in a probabilistic fashion. At regular intervals, a node calculates and exchanges its local load information with some randomly selected node. At all times, every node holds load information from a subset of nodes of fixed size. The load information about each node includes: processor speed, processor load, processor utilization and memory utilization. MOSIX measures processor load as the number of processes ready to run (in the ready queue).

As shown in Figure 4 process migration in MOSIX is accomplished in the following steps:

1. a source node detects a load imbalance
2. the source node selects a candidate process for migration
3. the candidate process selects a target node to migrate to
4. the process is temporarily suspended and its memory image and kernel state are transfered to the target node
5. the process is restarted at the target node

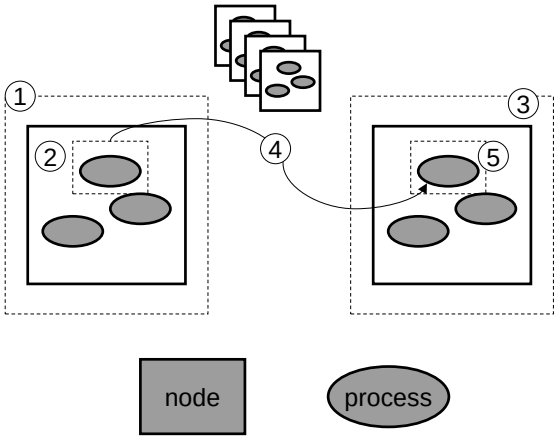

Figure 4 Mosix process migration steps

I will now describe each step in more detail.

Every node in a MOSIX cluster independently considers migrating one process out whenever new load information is received from another node and the new information reflects that there is some less loaded node in the cluster. This determination is made by procedure `load_balance` shown in Figure 5 called by the kernel at the source node. If a less loaded node is detected, `load_balance` selects the process p that has maximal migration priority as a candidate for migration. The migration priority is proportional to the process accumulated CPU time and the process contribution to local load. Processes that do not satisfy minimal conditions for migration are not considered. For instance, MOSIX does not migrate processes that have explicitly requested not to be migrated. The selected candidate process is marked and signaled for migration.

MOSIX delegates the responsibility to select a destination node to the candidate process. Once signaled, the process executes the `consider` procedure in Figure 5. The procedure computes a cost function $C[j]$ for each node j that the local node has load information on. The set of nodes considered is small (8 or so for a cluster of up to 512 nodes) and fixed. The cost function has three main components: expected CPU time to completion, communication overhead and migration time. Migration time is essentially proportional to process size, so it remains constant for the same candidate process (I don't understand why it has to be considered here). Each node keeps an execution profile for each local process that is used in calculating each component of the cost function. The node that minimizes the cost function is selected as the target. If the target node is

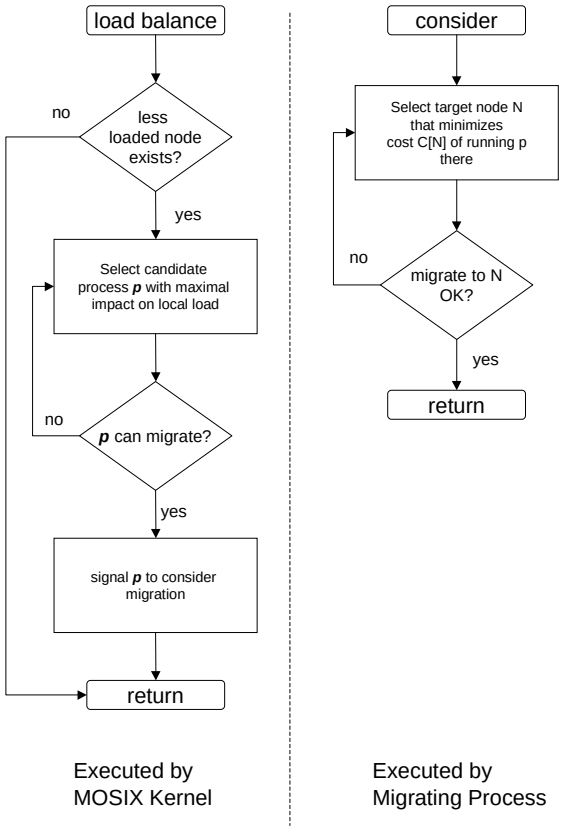

Figure 5 Load balancing procedures in MOSIX

the local node the process does not migrate.

During the next step, MOSIX copies the modified pages of the virtual memory image of the migrating process to the target node. Clean pages are brough in on demand from MOSIX's file system. MOSIX also transfers the process table entry.

The basic mechanism used to support transparent process migration in MOSIX is the kernel-to-kernel RPC. All location dependent system calls are processed in two steps. The first step, executed at the calling node, completes as much of the work as possible without relying on node location information. This first layer then makes an RPC call to an *ambassador process* running at the process' home node. The ambassador, a lightweight kernel process, completes the second step of the call by making a real system call at the home node. The result of this home node system call is channeled back to the calling process. Every system call executes locally or travels one hop to the process home node. Longer forwarding chains are not possible. Many UNIX system calls can be processed locally using state information transferred with the migrated process.

3.2 Virtues and Limitations

3.2.1 Transparency

Like GLUnix, MOSIX implements home node transparency. However, unlike GLUnix, MOSIX does does not expose location dependent environment information at the remote node.

3.2.2 Availability

Randomness in the information load information diffusion algorithm used by MOSIX achieves two main purposes: fault-tolerance and scalability. Fault-tolerance because the algorithm does not require to have information about all nodes. Scalability because the algorithm puts a bound on the number of nodes to which load information is transferred periodically. MOSIX's probabilistic algorithms avoid the complexity of consensus-based algorithms.

MOSIX does not rely in any centralized repository of configuration information. Therefore, it avoids the potential single point of failure that accompanies some centralized approaches.

3.2.3 Performance

A potential problem with MOSIX's load balancing approach results from nodes making load balancing decision independently using information that may vary from node to node. One possibility is that many nodes may decide to migrate processes to the same lightly loaded node, thus flooding it with too much load. MOSIX uses a technique that they call *export loads* to ameliorate this problem. The measurements that they present, however, do not measure the effectiveness of this technique.

In MOSIX, the unit of load balancing is the process. This approach seems appropriate when nodes execute a rich and time varying collection of processes from which to select candidates for migration. Such is not necessarily the case, for instance, when the cluster is dedicated to an internet information server. In this case, nodes run a fairly stable number of daemon processes. In this case it makes more sense to use requests as the unit of load balancing. A preemptive load balancing scheme using fully transparent process migration might not be the most cost effective way to provide load balancing.

Preemptive load balancing also seems to be appropriate when the cost of restarting a long running process is much higher that the cost of migrating the process. An alternative approach to preemptive load balancing may place the burden of migration on long running processes.

These processes are the most likely to need transparent migration. Each such *migration-aware* process may provide hooks that the kernel can use to signal the process that migration is about to happen. The process can be made responsible to perform any kind of application-level checkpointing necessary to minimize restart time. After checkpointing, the process simply dies. Migration simply entails restarting the process at the target node.

The high level of transparency in MOSIX's migration mechanism comes at a price in both performance and portability. The price in performance result from the extra communication overhead involved in forwarding some system calls to the process home node. As demonstrated by a number of attempts to provide user-level migration [16], this price seems unavoidable. The price in portability comes from the extra kernel support necessary to support transparent process migration. As a result commodity operating systems cannot be easily adapted to use MOSIX's load balancing features.

4 ParaStation

For completeness, I now describe the essentials of the ParaStation project. Since the project mostly deals with the fine-grain layer and the network layer, my discussion is rather schematic.

The main goal for ParaStation project was the design and implementation of fast networking hardware and software. The researchers wanted create a high performance massively parallel computer out of a cluster. The ParaStation designers focused their effors at the network and fine-grain layers of the layered cluster model. The fine-grain parallel layer is implemented directly on top of the network layer. The project designed and built a new network interface card and implemented a user level protocol stack. Processes syncronize their access to the protocol software using semaphores. The semaphores, however are implemnted at user level using processor intructions specialized for this purpose.

ParaStation implements several well-known parallel programming environements (e.g. PVM) on top of the user-level implementation of the BDS Unix sockets communication interface.

In ParaStation, workstations are connected to two networks, one via a fast propietary network interface card (NIC) and a second one via a standard (NIC). The user-level protocol stack is only available to applications that use the propietary network infrastructure. Users can develop parellel applications using a variety of parallel programming environments.

5 Microsoft Cluster Service (MSCS)

The Microsoft Cluster Service [24] is an extension to the Microsoft NT operating system. My discussion here will be limited to what the authors call the first developmental phase of MSCS. In this phase, MSCS is guaranteed to work well for clusters of two nodes. The effort has initially focused on providing the necessary support for making off-the-shelf *server* applications highly available. Future phases of MSCS will consider other issues including scalability to large numbers of nodes, load balancing and support for parallel applications.

MSCS provides two fundamental services: cluster membership and resource management. The cluster membership service automatically maintains a *view* of the currently active nodes in the cluster. All cluster members agree on the current view. MSCS notifies the active members of any changes in cluster membership. Changes in cluster membership may be caused by failures as well as by system administration actions. Upon noticing a membership change, all the nodes cooperate to maintain all the services provided by the previous membership.

While the cluster membership service provides detection and recovery from node failures, the resource management service provides detection and recovery from *software failures*. An MSCS *resource* is any hardware of software component that is necessary to provide some service. Examples of resources are disks, servers and IP numbers. *Resource dependencies* can be used to bundle resources together with other resources that they require to function properly. MSCS provide also provides allows resource groups to be defined in case some reason other that a direct dependency warrants grouping resources. Resource groups are the unit of recovery in MSCS. By grouping IP numbers with other resources providing the service, requests can automatically get forwarded to the node who provides the service.

MSCS resource management includes resource monitoring and resource failover/failback. At regular intervals MSCS polls the status of all its registered resources. If a resource appears to be down, MSCS attempts to restart it. If that doesn't work, MSCS automatically attempts to migrate the service, together with the rest of the resources in its resource group, to another active node.

The mechanism for persistent data failover in MSCS is disk switchover. MSCS requires that all the nodes in the clusters have physical access to the disks that contain data that should failover across different nodes. I will discuss the advantages and disadvantages of this approach to data failover in the "Virtues and Limitations" section below.

5.1 Implementation

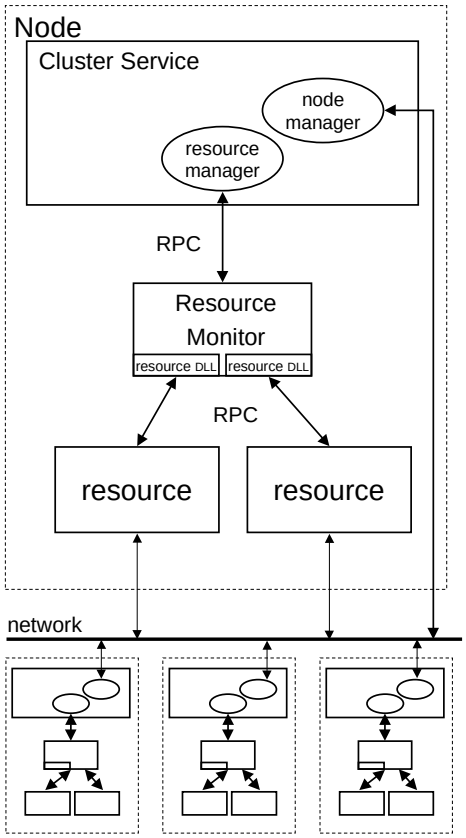

Figure 6 Organization of the Microsoft Cluster Ser

Figure 6 shows the architecture of the MSCS. T boxes in the picture correspond to processes and t ovals inside the boxes to process modules. The s vice consists a per-node process, the cluster service, a several resource monitor processes. The cluster serv implements most of the functionality of the MSCS. I source monitor processes serve as liaison between t cluster service and the actual resources. Communi tion between the resource monitors and resources done via dynamically linked resource control librar (RCLs).

In the remainder of this section I will discuss t following implementation issues in more detail:

- Maintaining a global configuration database
- Cluster membership algorithms
- Resource failover

Central to the MSCS design is maintenance of a configuration database that is replicated consistently across all active nodes in the cluster. The configuration database holds information about resources, resource groups and resources dependencies. It also maintains the current status of all nodes and all resources in the system. Any node can initiate a modification to the configuration database. Such modifications are implemented using atomic multicasts and are logged persistently in a special resource called the *quorum resource*.

The MSCS cluster membership algorithm tolerated both node failures and communication failures including network partitions. Detection of a failed node is done via a heartbeat mechanism [7]. Nodes periodically emit heartbeat signals to other nodes. A node that stops emitting heartbeats is assumed failed. Node failure detection triggers a *regroup* operation. This operation takes every node through several sequential rounds of messages as depicted in Figure 7.

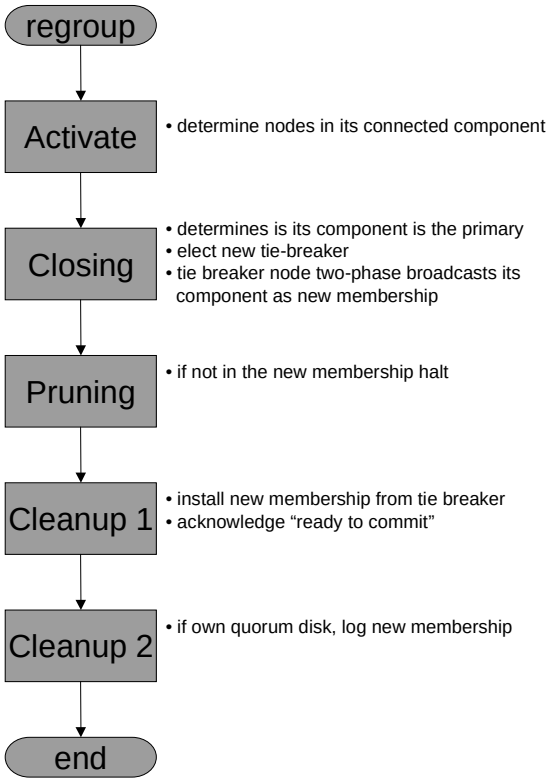

Figure 7 Message rounds in the MSCS regoup operation

The phases of the regroup operation are executed in

virtual synchrony [9]. Each node waits for every other node to finish the previous phase before moving on. During the Activate phase, each node calculates the set of nodes that it can communicate with. During the Closing phase each node determines if it belongs to the primary component (using a decision rule described below). During this phase a special node is elected as a new tie breaker (see below) and as a coordinator in an embedded two-phase commit of the new membership. The tie-breaker node informs all other nodes of the new membership. This message serves the purpose of the "prepare to commit" message in standard two-phase commit. During the Pruning phase all nodes that do not belong to the primary component shut down quietly. During the Cleanup 1 phase each nodes installs the new membership propagated by the tie-breaker and acknowledge "ready to commit". During Cleanup 2, the node that owns the quorum disk logs the new membership persistently.

Partitions can divide the cluster membership in one or more connected components. Allowing more that one component to provide service can generate a "split brain" situation in which each active component may attempt to provide service isolated from other components. The situation may lead to different components simultaneously modifying shared data potentially bringing such data to an inconsistent state. MSCS guarantees that only one component, the *primary*, can continue to provide service incorporating the following decision rule into the Closing phase of its regroup operation. A node knows its component is the primary if one of the following conditions hold. The conditions are listed in decreasing order of precedence.

1 the node component has a majority of the previous membership

2 the node component has half the previous membership, one of them is a tie breaker node and the number of members is at least two

3 the previous membership had two nodes, the number of nodes is one and the node owned the quorum disk in the previous membership

This rule shows the second use for the quorum resource as a tie breaker. A node will not attempt ownership of the quorum resource unless condition (3) above applies. Nodes that detect being outside of the primary component immediately shutdown.

After a regroup operation each node consults the configuration database and independently determines which resources it should own. These resources may include resources previously owned by nodes no longer in the current membership. Each node brings the resources it owns online. After this, the cluster is back in full operation.

5.2 Virtues and limitations

5.2.1 Transparency

MSCS was specially designed for ...

MSCS supports a weaker form of migration than MOSIX. Services are migrated by shutting down and restarting. This places the burden of making restart fast on the application. At the time that the application receives a shut down message, it can persistently save state to make restart faster. For the type of application that MSCS was designed, the server, this level of migration seems adequate.

5.2.2 Availability

MSCS achieves its goal of making off-the-shelf server applications fault-tolerant. The system simply provides a generic resource control library (RCL) that kills and restarts the server process in response to resource management events.

MSCS assumes that any persistent data managed by a highly available resource group be stored in a disk that is physically connected to every node where the resource group can migrate to. This approach to data failback presents a potential availability problem since every physically shared disk is a single point of failure for the cluster. The availability of the entire cluster is thus bounded by the availability of a disk subsystem. There are a number of techniques for making disks reliable (e.g. RAID) but this only moves our original availability problem inside the disk. We will come back to this point in section 7 when we will discuss an alternative design for MSCS that uses replication as the main mechanism for highly available persistent storage for both configuration and application data.

MSCS dependence on a single quorum resource is another potential source of both availability as well as of performance bottleneck problems. The quorum disk serves two purposes, each of which could be served by any other node in the cluster. First, the quorum resource logs changes to the configuration database. There is no reason (except storage overhead) why each node could not log these changes. The second role for the quorum resource is to serve as a tie breaker in the specific case when a primary component of two nodes partitions and each nodes attempts to become the new primary component. The elected tie-breaker node could be used in this case.

5.2.3 Performance

MSCS uses a simple write-all [8] approach to maintain the configuration database consistently replicated at every node. In this approach, the cost of reads are low and independent of the number of nodes. The cost of writes, however, is high since every node must be contacted to update its copy. Although the configuration database only changes when components fail, for clusters of many nodes this cost may be prohibitive. The rate of node failures also increases linearly with the number of nodes. For large clusters, maintaining the replicated data with voting [13] [22] [1] may reduce the cost of writes at the expense of increasing the cost of reads.

6 Summary and Comparison

The table in Figure 1 summarizes the achievements of each system in each of the comparison criteria of interest.

With respect to transparency, both GLUnix and MOSIX achieve home node transparency, but MOSIX provides a much stronger level as a result of its forwarding of location dependent system calls directly to the home node. MSCS on the other hand, provides a completely different type of transparency. To clients of an MSCS service, the cluster looks like a single server. ParaStation supports the fine-grain parallel abstraction.

In terms of availability, both Berkeley NOW GLUNix and MSCS have single points of failure. While GLUnix cannot tolerate a single failure of its master process, MSCS cannot tolerate a single failure of its quorum resource nor of its shared disks. Only MSCS provides automatic application failover and restart. This results in a reduction in mean time to repair (MTTR). MSCS can maintain replicated data consistent across network partitions. MOSIX can also tolerate partitions due to its symmetric and probabilistic load diffusion algorithm.

GLUnix has one potential performance bottleneck in its master daemon process. MSCS has potential performance bottlenecks in its quorum resource and on its shared disk bus. Both MOSIX and GLUnix offer some sort of load balancing. ParaStation offers improved performance with its user level implementation of network protocols as well as with fast network interface card (NIC) hardware.

7 An Alternative MSCS design using replication

This section presents an alternative design for the Microsoft Cluster Service. Central to the new design is the use of replication algorithms based in voting [13] [22] to provide a general replicated object service that can be used to maintain the MSCS configuration database consistent and fully replicated. The design leverages as much as possible on services already available from MSCS.

7.1 Motivation

A number of factors have influenced my decision to redesign MSCS. First, MSCS uses different complex protocols to solve rather similar problems. As was first demonstrated in [1], both the global update protocol and the regroup protocol could be unified in a single simple protocol built on top of a transactional storage system. The second factor is that although MSCS includes all the algorithms necessary to implement a highly-available replication service, it hides the service inside the cluster service. Other applications do not have access to the replicated data service. Finally, replication removes the dependency of MSCS on disk switchover. As I mentioned before, lack of support for replication in MSCS forces disks to be physically shared and become single points of failure.

7.2 System design

First, the design proposes the creation of a new resource group consisting of three resources.

replication disk resource Used as for storing copies of replicated data

distributed transaction resource provides a distributed transaction service across all nodes

replicated data resource Implements a transactional replicated data storage service

These resources are never migrated and are monitored using the failure management techniques (e.g. RCLs) provided by MSCS.

The replication disk can simply be an instance of the standard disk resource type, but it must be dedicated to the replication service. The distributed transaction resource uses classical distributed transaction techniques to present an interface through which clients can submit sequences of operations on objects that will be carried out atomically. Transactions submitted to the distributed transaction resource may include operations on remote objects, but references for such objects must include all the information necessary to locate them.

Objects and object copies are located by the replicated data resource by keeping replicated directories. In addition to the normal information kept in NT directories, replicated data resource directories include information necessary to locate multiple copies objects

System	Transparency	Availability	Performance
Berkeley NOW	home node limited	single point of failure no fail-over	load balancing bottleneck
MOSIX	home node transparent	masks failures no fail-over tolerates partitions	load balancing low msg overhead
MSCS	server	single point of failure low MTTR tolerates partitions	bottleneck
ParaStation	MPP	no support	user level protocol stack fast NIC hardware

Figure 8 Transparency, availability and performance achieved by each system

among the different transaction services. The replicated data resource service translates replicated data resource (logical) operations on objects (reads and writes) into transactions holding operations on *copies* of objects.

Creation of each new resource entails the implementation of corresponding resource control libraries. The distributed transaction resource must implement the standard transactional recovery procedures whenever it is brought online.

Figure 9 shows the architecture of the new replication-based MSCS. Note in particular how the cluster membership service is now built on top of the replicated storage service. The replicated storage service communicates with the distributed transaction service. Both resources are managed by the MSCS resource manager via resource control libraries. Since all the resources in the new resource group are fixed and supported at all nodes, their configuration information is essentially fixed and does not require access to the configuration data kept by the services themselves.

7.3 Virtues and Limitations

7.3.1 Transparency

The alternative design provides exactly the same level of transparency as the original design.

7.3.2 Availability

The new design removes the dependency of MSCS on switchover and can tolerate a number of failures on the replication disks proportional to the level of replication maintained. This has the potential to improve availability substantially.

7.3.3 Performance

The new design also improves performance by eliminating the potential performance and scalability bottleneck of the shared disk bus.

In the remainder of the paper I will focus my attention on a problem that hasn't been dealt with optimally by any of the systems discussed so far: data failover/failback.

8 Adaptive Replication for Data Failover-Failback

Ideally, when a cluster node fails, the remaining nodes should transparently take over the responsibilities of the failed node. The system as a whole should not fail, but rather should recover and continue operating at the maximum capacity possible. This requires that the data accessible to the failed node be made accessible to the node or nodes taking over its duties. The process of making data from a failed node available to other nodes is called *data failover*. When the failed node recovers it should get its (possibly updated) database from the nodes that replaced it. This process is called *data failback*.

A process similar to failback is also applied when nodes are added to the cluster to increase system capacity. In this case data can be reorganized and the new node is assigned some portion of the data.

There are two possible ways in which data failover/failback can be implemented: *switchover* and *replication*. I will describe each technique and argue how replication can suit a cluster architecture better than switchover. For the purposes of this explanation I will call the failed node *source* and the replacing node the *target*.

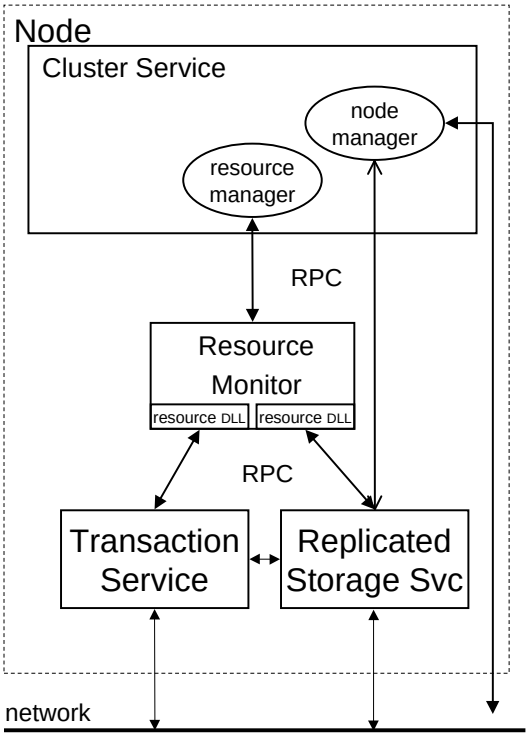

Figure 9 Alternative Design for MSCS based on re

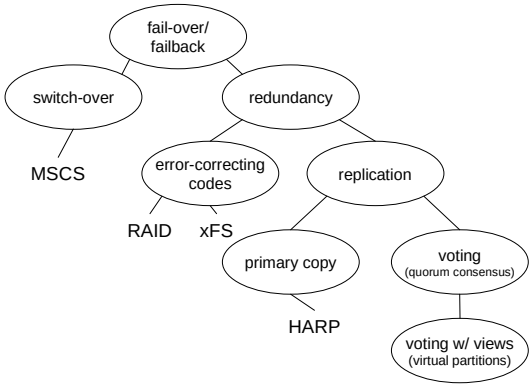

Figure 10 Hierarchy of approaches to data failover and failba

Swithcover requires that the source and target no physically share disks. During normal operation all the nodes work on disjoint subsets of data. As a result, every single shared disk is a single point of failure. If it can be assumed that all disks fail independently, as the number of shared disks increases the situation worsens since the failure rate increases linearly with the number of disks. The net decrease in availability could be substantial.

An alternative to swithcover is *replication*. Replication simply stores multiple copies of objects across different disks. When a node fails the object remains available at some other node. As long a there is a way to direct operations to the nodes that have copies of the relevant objects available, the system can continue to offer service.

An interesting problem results when one considers how a cluster with replicated data should react to the addition of a new node. In order to take advantage of the increase in storage capacity the cluster software has two basic alternatives. It can store additional copies of objects in the new node (Figure 11) or it can migrate (Figure 12) some copies of objects to the new node.

If objects are very seldom written and most oftenly read, the replicating is always an acceptable alternative. This is due to the fact that many replication algorithms, including voting and primary copy [2] are capable of implementing much faster writes than reads. The more interesting case is when the frequency of writes is large enough that the increase in the cost on write operations is not acceptable. In this case, migration should be considered. But why not just use migration? In this section I present arguments as to why relying exclusively on migration will decreases availability.

A distributed storage system is said to be *adaptively replicated* it incorporates algorithms that automate the choice between migration and replication in order to maintain a desired level of availability in the presence to changes to the number of nodes. I know of no system that incorporates such an algorithm. There have been distributed file systems that provide a level of replication that is fixed at system configuration time [15]. Others provide facilities to create objects with different levels of replication have been implemented in the past [13]. Deceit [21] provides operations to dynamically alter the level of replication of a file. However, none of these systems automatically alters the level of replication in response to a change in the number of nodes.

Adaptive replication requires solving two problems: (1) Deciding when to replicate and when to migrate, and (2) carrying out the operations consistently in the presence of failures. For this paper I will focus on (1). I will accept the existence of systems that allow dynamic changes to the level of replication of objects as convincing evidence that (2) is achievable with known technology.

I have three hypotheses:

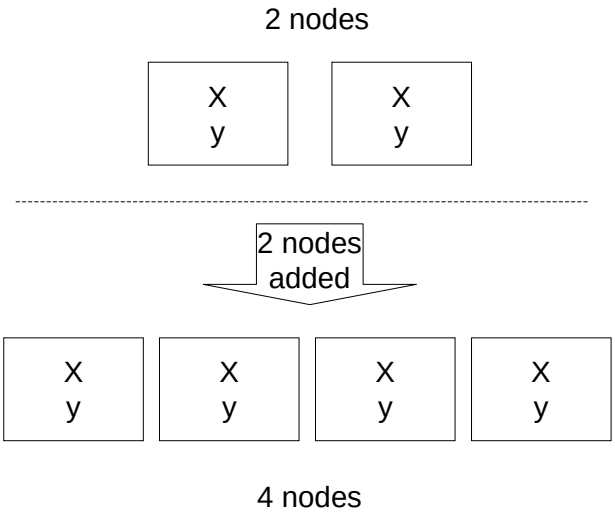

Figure 11 Replication increases the number of copies

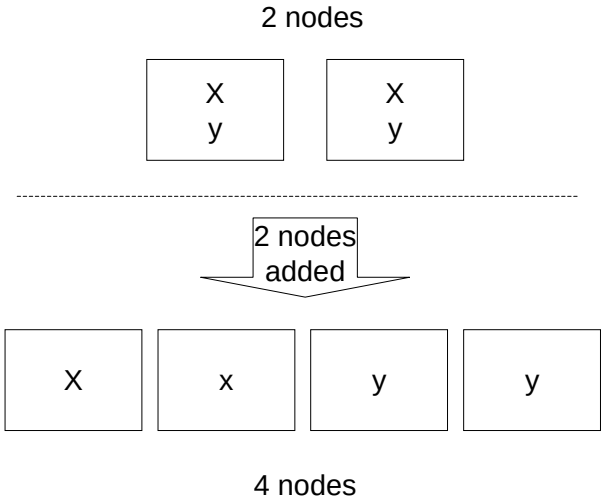

Figure 12 Replication distributes existing copies

H1 I a system relies exclusively on migration as a way to use the storage capacity contributed by a new node, its availability will decrease.

H2 A system must alternate migration with replication if it wishes to maintaining a desired level of availability at minimal cost

H3 Replication should be applied much less often than migration

H2 is not quite independent form the others. It follows from H1 and the fact that replication increases operation costs.

I will now argue that H1 and H3 are true under the following simplifying assumptions about a system with n nodes.

A1 system keeps same number k of copies of each object

A2 Initially $n = k$

A3 n increases k nodes at a time

A4 no network partitions occur

Under these assumptions a system of n nodes keeping k copies per object can always be represented in matrix form as shown in Figure 13. I such a system objects cab be partitioned by a relation that associates objects that have copies on exactly the same nodes. The number q represents the number of equivalence classes in such partition.

The argument is twofold. First I argue that, if only migration is used, the maximal availability attained by a system of q groups after adding k nodes is obtained by partitioning the objects in $q+1$ groups where the set of nodes where each group has copies is disjoint with that of any other group. The reason why this is true is that in such a system each failure can only touch one group. The probability that k nodes of the same group will fail, with the consequent total failure, is smaller that when the number of groups touched by each failure increases. But this must be the case when the sets of nodes of different groups overlap.

The second part of the argument is easier to see. Notice that a total failure occurs when all the nodes in one row fail. The probability that this happens is $q * p^k$ where p is the probability that each node fails. Therefore the availability of the system is one minus this probability:

$$A(k, q) = 1 - q * p^k \tag{1}$$

This result demonstrates that a system relying exclusively on migration will suffer from steady decreases in availability. It remains to be shown whether this decrease in availability is significant for real systems. Also, I must determine the impact that relaxing the simplifying assumptions will have on this result.

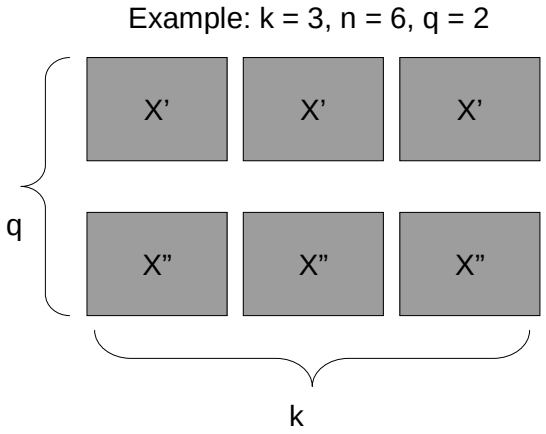

Figure 13 A system of n nodes and k copies of each object

9 Conclusion

I presented an analysis of four approaches to provide various system abstractions on clusters of workstations. The approaches have been evaluated using trasparency, availability and performance as criteria. My most important comtributions are:

- Proposed an alternative simpler design for the Microsoft Cluster Service that should achieve both higher availability and higher performance
- Proposed the idea of adaptive replication, maintaining a desired level of availability by automatically adjusting replication level in response to changes in the configuration if the cluster
- Provided evidence of the need to support adaptive replication in clusters

Further development of the idea of adaptive replication is necessary along the following lines:

- Must show that the rate of change of change in availability as a consequency of migration is fast enough to warrant adaptive replication in real clusters
- Formal proofs of the arguments provided to support the various hypotheses

References

[1] Amr El Abbadi and Sam Toueg. Availability in a partitioned replicated database. In *Proceedings of the Fifth ACM Symposium on Principles of Database Systems*, pages 240–251, 1986.

[2] Peter A. Alsberg and John D. Day. A principle of resilient sharing of distributed resources. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 627–644, October 1976.

[3] Thomas E. Anderson, David E. Culler, and David A. Patterson. The case for now (networks of workstations). *IEEE Micro Magazine*, February 1995.

[4] Thomas E. Anderson, Michael D. Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Y. Wang. Serverless network file systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, pages 15–28, December 1993.

[5] Amnon Barak, Shai Guday, and Richard G. Wheeler. *The MOSIX Distributed Operating System.* Lecture Notes in Computer Science. Springer-Verlag, 1993.

[6] Amnon Barak and Oren La'adan. The mosix multicompuer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(45):361–372, March 1998.

[7] Joel Bartlett. A non-stop kernel. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles*, pages 22–29, December 1981.

[8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1987.

[9] Kenneth P. Birman and Robbert van Rennesse. *Reliable Distributed Computing with the Isis Toolkit*, chapter 5, pages 79–100. IEEE Computer Society Press, Los Alamitos, California, 1993.

[10] Fred Douglis and John K. Ousterhout. Trasparent process migration: Design alternatives and the sprite implementation. *Software Practice and Experience*, 21(8):757–85, August 1991.

[11] Al Geist, Adam Beguelin, Jack Dongarra, Weincheng Jiang, Robert Manchek, and Vaidy Sunderam. *PVM: Parallel Virtual Machine - A User's Guide and Tutorial for Networked Parallel Computing.* MIT Press, Cambridge, Massachusetts, 1998.

[12] Douglas P. Ghormley, David Petrou, Steven H. Rodrigues, Amin M. Vahdat, and Thomas E. Anderson. Glunix: a global layer unix for a network of workstations. *Software: Practice and Experience*, 28(9):929–961, July 1998.

[13] David K. Gifford. Weighted voting for replicated data. In *Proc. of the 7th Symp. on Operating Systems Principles*, pages 150–162. ACM, December 1979.

[14] John H. Hartman and John K. Ousterhout. The zebra striped network file sytem. *ACM Transactions on Computer Systems*, August 1995.

[15] Barbara Liskov, Sanjay Ghemawat, Robert Gruber, Paul Johnson, Liuba Shrira, and Michael Williams. Replication in the harp file system. In *Proc. of the 10th Symp. on Operating System Principles*, 1988.

[16] Michael Litzkow and Marvin Solomon. Supporting checkpointing and process migration outside the unix kernel. In *USENIX Association 1992 Winter Conference Proceedings*, pages 283–290, December 1992.

[17] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer Magazine*, 23(5), 1990.

[18] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The sprite network operating system. *IEEE Computer Magazine*, 21(2), 1988.

[19] Gregory F. Pfister. *In Search of Clusters: The Ongoing Battle in Lowly Parallel Computing.* Prentice-Hall PTR, New Jersey, 1998.

[20] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, October 1991.

[21] Alex Siegel, Kenneth Birman, and Keith Marzullo. Deceit: A flexible distributed file system. Technical Report TR89-1042, Cornell University, Department of Computer Science, 1989.

[22] Dale Skeen, Amr El Abbadi, and Flaviu Cristian. An efficient fault-tolerant protocol for replicated data management. In *Proceedings of the Fourth ACM Symposium on Principles of Database Systems*, pages 215–229, 1985.

[23] Supercomputing Technologies Group, MIT-Laboratory for Computer Science. *Cylk 5.2 (Beta 1) Reference Manual*, July 1998. http://www.supertech.lcs.mit.edu.

[24] Werner Vogels, Dan Dumitriu, Ken Birman, Rod Gamache, Mike Massa, Rob Short, John Vert, Joe Barrera, and Jim Gray. The design and architecture of the microsoft cluster service: A practical approach to high-availability and scalability. In *Proceedings of the 28th symposium on Fault-tolerant Computing*, Munich, Germany, June 1998.

[25] Thorsten von Eicken, David E. Culler, Seth Copen Goldstein, and Klaus Erik Schauser. Active messages: A mechanism for integrated communication and computation. In *Proceedings of the 19th ACM International Symposium on Computer Architecture*, Gold Coast, Australia, May 1992.

[26] Thomas M. Warschko, Joachim M. Blum, and Walter F. Tichy. The parastation project: Using workstations as building blocks for parallel computing. In *Proceedings of the International Conference on Parallel and Distributed Processing (PDPTA '96), Techniques and Applications*, volume I, pages 375–386, Sunnyvale, California, August 1996.