Using the ISIS Resource Manager for Distributed, Fault-Tolerant Computing*

Timothy Clark
Kenneth Birman

TR 92-1289
June 1992

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under DARPA/NASA subcontract NAG 2-593 administered by the NASA Ames Research Center and by grants from GTE, IBM, and Siemens, Inc. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy or decision.

Using the ISIS Resource Manager for Distributed, Fault-Tolerant Computing*

Timothy Clark
Kenneth Birman
Department of Computer Science, Cornell University

June 23, 1992

Abstract

Under current versions of the UNIX^{tm} operating system, it is difficult to take advantage of the massive computing power of idle or lightly-loaded workstations on a network. This paper introduces the ISIS Resource Manager, a distributed, fault-tolerant application capable of recapturing this processing power, as well as providing a transparent interface to network computing resources.

1 Introduction

Networks of inexpensive UNIX^{tm}based workstations offer great promise as computational engines for a wide range of coarsely parallel applications. Unfortunately, under contemporary versions of the UNIX operating system, effective utilization of multiple machines can be clumsy at the shell level, and impractical at a program level.

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under DARPA/NASA subcontract NAG2-593 administered by the NASA Ames Research Center, and by grants from GTE, IBM, and Siemens, Inc. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision.

In this paper, a utility that solves this problem is described. The ISIS resource manager gathers collections of heterogenous, idle workstations into a processor pool onto which tasks are scheduled, and deals with dynamic events such as crashes, the need to release a workstation when its owner resumes active use, and the introduction of new types of machines and software services while the system is running. The solution is easy to use, has been ported to a wide variety of UNIX platforms, and offers multiple interfaces oriented towards different classes of users. One of these mimics a traditional batch queueing system, while others are more dynamic and suitable for use in explicitly distributed software systems that exploit program and data replication to increase performance or fault-tolerance.

The resource manager has been applied to problems in computer aided design, simulation, scientific computing, distributed software development, graphics and many other areas. These uses include existing applications that must be executed without modification as well as new software that benefits by making explicit use of the resource manager at the program level.

This paper is structured into three sections: the first section discusses the growing importance of high-speed workstation networks and the need for network management utilities; the second section discusses how advances in software technology have made it possible to easily construct a distributed, fault-tolerant application to meet this need; and the final section describes the architecture of the resource manager, as an example of this class of application.

2 The Growing Importance of Networks and Coarsely Parallel Software

As networks have become increasingly prevalent, more and more users are encountering problems that can benefit from being run on multiple machines in parallel. As one example, VLSI circuit simulation typically involves many trial runs of a circuit using different combinations of inputs. Here, a single program is run many times with different inputs. Such a problem is ideal for solution using a network of conventional high-speed workstations. Indeed, since the communication requirements of such an application are minimal, a closely coupled multi-processor is not needed, and the workstation approach will be considerably more efficient than batch style execution on a shared supercomputer. This trend has created a need for automatic network resource management tools.

Unfortunately, under contemporary versions of the UNIX operating system, effective utilization of collections of machines can be awkward. The available tools include programs such as `ruptime`, `rwho`, `rsh` and `rlogin`. Using these, it is difficult to determine which machines are the most appropriate ones to use, and there is no network-wide scheduling mechanism to enforce fairness. There are no tools to check the status of active programs on the network, or to prioritize the use of certain machines in favor of certain sets of users. If an application may run for hours, days, or even weeks, and must be automatically monitored and restarted in the event of failure, there may be no human in the loop at the time a network scheduling activity is required. UNIX provides little help in these areas, forcing users to cobble together approximate solutions using `mail` to report completion status, periodically running `ps` to check on job activity, and so forth.

The resource manager solves these problems by providing an easy to use, portable, fault-tolerant mechanism for job management and monitoring in a distributed environment.

3 Advances in Software Technology Allow for Robust Applications

Much research has been done on the prospect of exploiting the power of distributed networks of workstations. However, most research provides only several pieces of the puzzle, not the whole solution. Shared memory models, reliable broadcast protocols, remote procedure calls, etc.. all contribute to the ability to distribute an application across the network. However, the development of the ISIS Distributed Toolkit technology addresses the broader, more complete picture.

Basically, ISIS is a subroutine package that employs protocols built over UDP to ensure that messages will be delivered reliably and in order; it adds headers to messages and delays messages on arrival (if necessary) to accomplish this. This reliable, consistent ordering of messages is called "Virtual Synchrony". Events such as multicast and detection of failures are atomic in a virtually synchronous setting; events appear to happen *one-at-a-time* and in a consistent order at all sites.

ISIS provides a rich set of distributed programming techniques based on these protocols. Central to ISIS is the notion of a *process group.* These groups are a lightweight programming construct: a single process can belong to arbitrarily many groups, and there is minimal overhead in being a member of a group. A process can dynamically join and leave groups, and groups can span multiple machines. ISIS provides a state transfer utility and several multicast and unicast communication primitives, with differing levels of ordering guarantees, for point-to-point and group communication. A multicast can be directed to all members of a group, and zero or more will respond, depending on the needs of the particular application.

Although the overhead of using a package such as ISIS may negatively impact some applications such as real-time systems, its effect on an application such as the resource manager is negligible. In the resource manager, the time to process a job request is heavily biased by the fork/exec system calls. The communication and ordering overhead of ISIS represents only about 20ms of the 450ms average response time for a job request on a SUN sparcstation1.

All three pieces of the resource manager system are built upon the ISIS Distributed Toolkit, and use causal and atomic multicast, process group communication, data replication and failure detection. The use of ISIS relieved much of the difficulty associated with failures, synchronization, consistency and network communications. Taken together with the wide range of tools represented in the toolkit, this approach led to major improvements in the robustness of the system. Unfortunately, space limitations on this paper preclude a more detailed discussion of the ISIS protocols and concepts. More information on ISIS, and performance data, can be found in [BJ87a, BJ87b, BJKS88].

4 Architecture of the Resource Manager

The resource manager system consists of three parts: the resource manager server; the resource manager stubs; and the various user-interfaces. The resource manager server normally runs on 2 to 5 machines, while the stub program runs on all machines which are to be included in the resource pool. The user-interface programs run on end-users' workstations. See Figure 1.

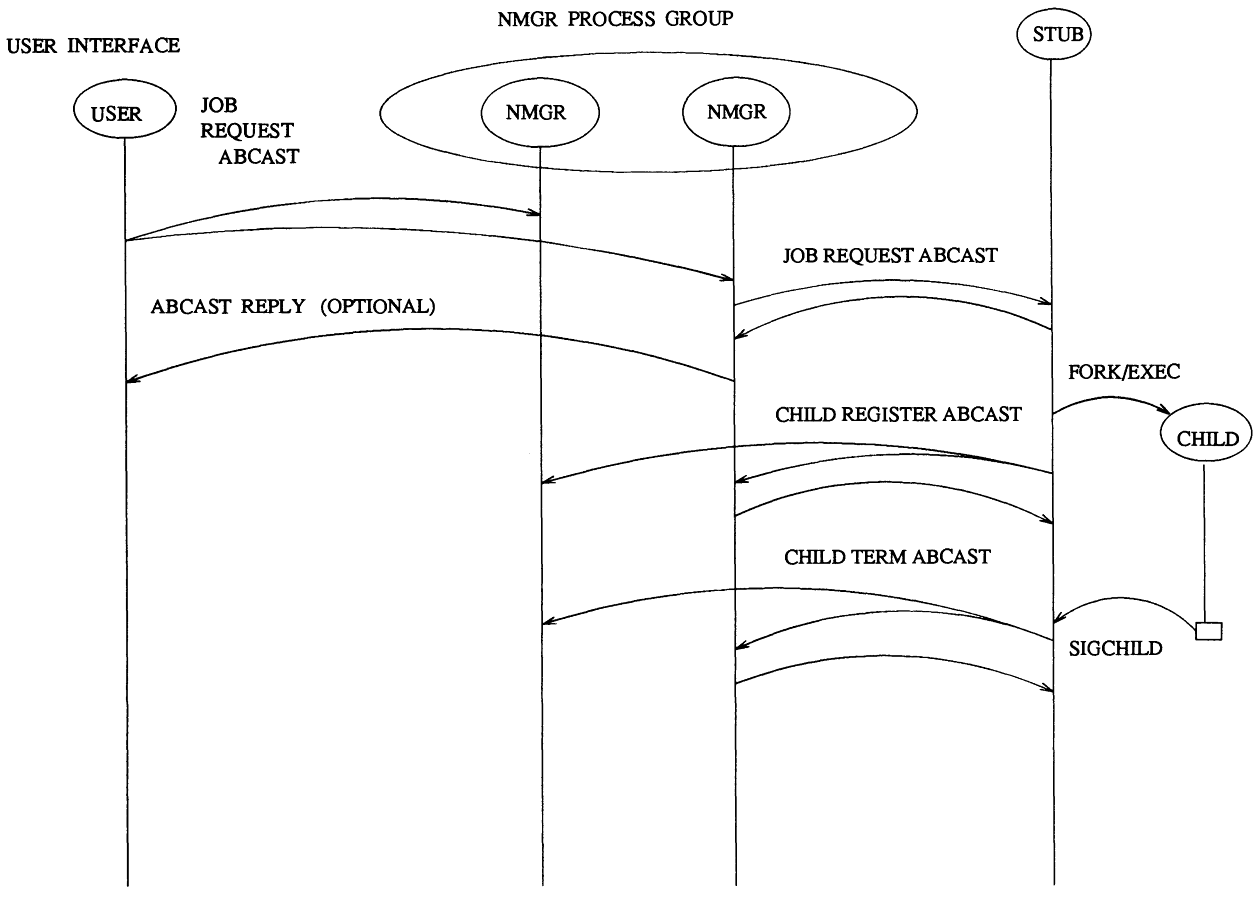

Figure 1: Architecture of the ISIS Resouce Manager

4.1 The Resource Manager Server

The resource manager server manages a customizable database of registered machines and services. Normally several copies of the resource manager server are instantiated, forming a process group with fully replicated data for fault-tolerance. All communication between the remote stubs, the resource manager server process group, and the user-interfaces use the ISIS reliable multicast protocols, cbcast and abcast. Using the ordering guarantees of these protocols, every event is seen in the same order at each server, so if a server or stub crash, each remaining server has identical, consistent information upon which to act.

Each resource manager server executes the same code in parallel and responds to incoming events by using the ISIS lightweight tasking subsystem to fork off tasks as event handlers. The event types recognized by the resource manager include:

- resource manager process group join or leave event
- resource manager stub join or leave event
- resource manager stub load update message
- user-interface request message
- API request message
- child (job) registration message
- child (job) termination message

Each resource manager server is sensitive to server and stub join/leave events. Upon receiving a server join request, the oldest member of the server process group initiates a state transfer to the joining server. All events are briefly suspended while the transfer takes place, insuring that all database information is received by the joining member in a consistent manner. Upon a server leave or failure event, the surviving members of the server group reconfigure, and any stubs connected through the failed server reconnect to a surviving server.

The resource manager implements user-definable, priority-based queues, which mimic traditional batch queueing mechanisms, but which provide the *fault-tolerant* execution of jobs. Upon a stub join event, the servers signal the batch queueing mechanism that a new machine has joined the resource pool, and search for any job which matches the new machine's specifications. Upon a stub leave event, the servers search their database for any jobs which had been running on that stub. If jobs are found, they attempt to restart them on another machine before deleting the failed stub's information from their database. The jobs from the failed stub will either be started on a different stub, or will be placed in a queue until a matching stub machine becomes available.

When a job request is received, a newly created task will add the job-related information to the resource manager's database and attempt to find a machine on which to execute the job. A pattern matching algorithm is run in combination with a simple, but effective load-balancing algorithm to select a machine on which to run a submitted job request. If a matching stub machine is found, a message is sent to the stub containing all the information necessary to execute the job. If no matching machine is found, the job is placed in a batch queue until a matching machine becomes available.

Each stub is responsible for sending in a CHILD_REGISTER and CHILD_TERM message for each job it starts. The CHILD_REGISTER message contains the remote stub's machine name and the new job's process id and informs the server that the job was started successfully. The CHILD_TERM message returns the job's exit status and signals the normal or abnormal completion of the job. Upon receiving a CHILD_TERM message, the resource manager may signal the batch queueing mechanism that the stub is available to run another job. Depending on the job specifications and the exit status, the server may take other actions as well, such as automatically restarting the job or sending mail to the initiator of the job.

All machine and job related information is stored in a replicated database by each server and is available for status queries via one of the user-interface programs.

Below is an example of a resource manager database for a small network which specifies key words, machine types, individual machines, a batch queue, and an administrator's login id (for email notification upon failure events).

```
key   fpu     /* Has hardware fpu */
key   mathlab /* Has mathlab license */
key   xfpu    /* Has high speed FPU support */
```

```
key   sparc   /* Sparc architecture */
key   mc68020 /* MC68020 architecture */
key   mips    /* mips architecture */
mtype sun3    = {mc68020,fpu, mem=8, rating=1.0}
mtype sparc1+ = {sparc, mem=16, rating=8.0}
mtype sparc2  = {sparc, fpu, xfpu, mem=16, rating=16.0}
mtype mips    = {mips, fpu, xfpu, mem=16, rating=16.0}
mtype hpux    = {hpux, fpu, xfpu, mem=16, rating=16.0}

machine flute = sparc1+
machine viola = {mips, mathlab,mounts={/usr/fsys/fs1,/usr/fsys/fs2}}
machine bongo = {sparc1+,mem=32,mounts={/usr/fsys/fs1}}

add_q NEW_Q = {nq_maxsize=120,nq_maxactive=100,nq_maxrun=200,\
               nq_maxmem=20}
admin_id  tclark
```

The resource manager recognizes many options within job specifications for services such as:

- sending mail upon completion of a job
- restricting the number of jobs simultaneously running on a machine
- automatically restarting reliable server-based applications
- scheduling cron jobs or sequentially-related jobs
- requesting specific machines for execution
- copying files into and/or out of the execution directory
- arbitrary, user-defined attributes used to match specifically- equipped stub machines to job requests, eg. machines licensed to run specific software

4.2 The Resource Manager Stub

The resource manger stub is run on each workstation which wishes to participate in the resource pool. When this program is instantiated, it first reads in its machine-specific information from an initialization file and then registers with the resource manager server, communicating its host name, machine type, current load, and mounted file systems. Once registered, its only activity is to send in its current load information at regular intervals and await incoming job requests from the server. In this state, the cpu overhead of running the stub on a workstation is insignificant.

When a job request is received by a stub, the message is unpacked, retrieving the binary name to be run, the arguments, and the environment variables. It will then use the UNIX fork/exec system calls to execute the requested binary. The fork/exec system calls represent the major portion of the system's performance, the actual communication of the job request taking approximately 20 ms. After the fork/exec, the stub sends in a CHILD_REGISTER message to the resource manager group, containing the process id of the spawned job. The UNIX wait() system call is used to reap terminated child processes and their associated exit status, and this information is passed on to the resource manager server(s) via a CHILD_TERM message.

The stub program is highly configurable. Machine-specific information is associated with the stub through the initialization file. This file contains attributes such as the machine type, the amount of physical memory, the "rating" of its processor (in user-definable units), the number of processors available (for multi-processors), and other user-definable attributes such as software licenses or special hardware options (eg. floating point processors).

The stub's command line arguments allow the specification of the following behavioral options:

- go to sleep for a specified time interval if console login takes place
- go to sleep if keyboard/mouse activity occurs on the host console
- go to sleep if the host cpu load goes beyond a specified threshold level
- the ability to "plug-in" a user-defined load monitor routine
- the specification of the execution directory where job requests will be run (the default is /tmp/nmgr)

4.3 The Resource Manager User-Interface Programs

The resource manager has three types of user-interfaces: a shell-level command line interface; an X Windows/Motif graphical user-interface; and an applications programmer interface (API). The typical response time for a completed job request (via netexec or xnmgr) is on the order of 1/2 second.

The Shell-Level Interface

A shell-level interface has been implemented using the names **`netexec`**, **`netkill`**, **`netstatus`**, **`reserve`** and **`unreserve`** - all links to the same binary. These interfaces allow the user to start, monitor, and kill jobs, as well as the ability to "reserve" machines for future use and "unreserve" them, as desired, in a transparent manner.

```
Example 1:

net_exec "{binary={sparc=/usr/u/fred/test},min=3,mem>10,\
          send_mail}"
```

This job request will start 3 copies of the binary /usr/u/fred/test on sparc-based workstations with physical memory greater than 10 MB, and will send email to the user upon job completion.

```
Example 2:

netexec "test1={binary={sparc=/usr/u/fred/sparc/test,\
                mips=/usr/u/fred/mips/test},args={4},\
                min=4, mounts=/usr/u/fred/data}"
```

This job specification will execute four copies of the binaries /usr/u/fred/sparc/test or /usr/u/fred/mips/test with the argument "4" on sparc or mips-based workstations that have the /usr/u/fred/data file system mounted. The job will be run under the job name "test1", which can be used by the `netkill` program to kill the job(s) if so desired.

```
Example 3:

netstatus -j

will output:

Job Name       Binary Name            Where
test1.tclark  /usr/u/fred/sparc/test  flute.cs.cornell.edu
test1.tclark  /usr/u/fred/mips/test   viola.cs.cornell.edu
test1.tclark  /usr/u/fred/sparc/test  1 instances queued
test1.tclark  /usr/u/fred/mips/test   1 instances queued
```

XNMGR - the X Windows/Motif Graphical User-Interface

The graphical interface, based on the Motif widget set, is called xnmgr. This interface, using popup menus, push buttons, sliders and text entry widgets, provides full functionality access to the resource manager, including the management of jobs (start, suspend, restart, kill), the creation of batch queues, and more detailed status information. See Figure 2 for a depiction of the Job Submission screen.

The Application Programmer Interface (API)

An API was added to the resource manager to provide the ability to communicate at the program level. The API provides an entry point for generic network manager commands, such as adding machine or service types or making job or status requests.

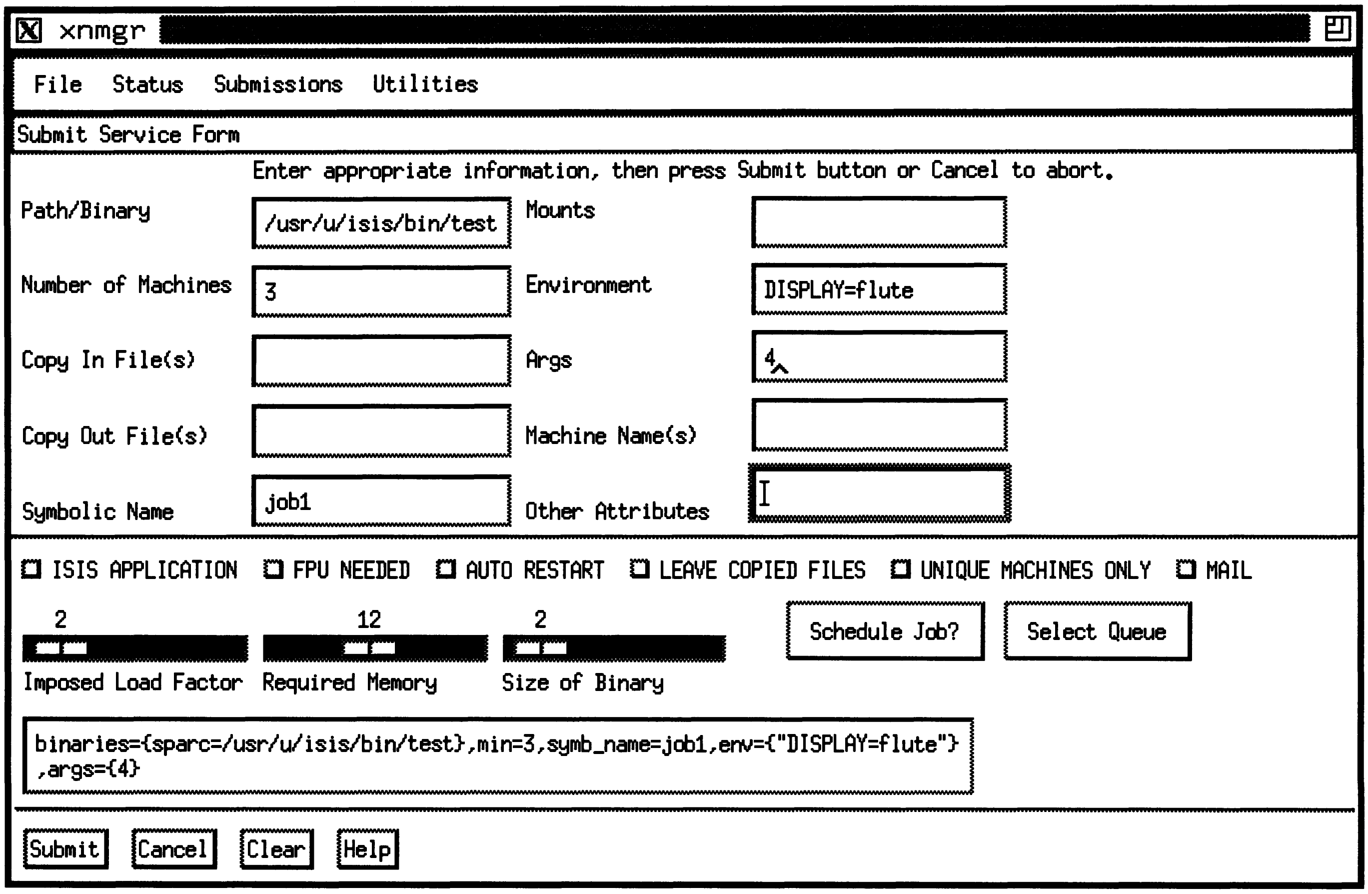

Figure 2: XNMGR Job Submission Screen

5 Conclusions

The evolution of computing networks has moved through a series of stages. In the 1980's, workstations were relatively new on the scene and represented a huge amount of computing power with respect to time-shared processors. Applications needing still more power generally turned to mainframes, special purpose mini-supercomputers and supercomputers. It is only recently that networks have become so common and workstations so fast that the idle processing power of a typical network routinely exceeds the processing capacity of mid- range supercomputing systems. Indeed, within a few years, the aggregate capacity of a network of high-end workstations will outperform all but the fastest supercomputers. Utilities like the ISIS resource manager place this huge computational resource at the disposal of non-expert UNIX programmers, permitting a style of cooperation and sharing that would previously have been difficult and impractical. Moreover, the resource manager can play a valuable role in applications that require high reliability and fault-tolerance, or wish to exploit parallelism for improved response time.

The resource manager is also interesting as an illustration of how new technology, such as the ISIS programming environment, can provide the necessary tools to allow the relatively easy development of a reliable, distributed application. Using ISIS permits one to focus on the functionality of the subsystem rather than the intricate and subtle problems raised by network protocols and fault-tolerance.

Looking to the future, it seems reasonable to predict that a new wave of high reliability, easily employed distributed programming tools will transform the UNIX networks of the 90's into much more flexible and highly integrated distributed computing environments than are presently available.

References

[BJ87a] Kenneth P. Birman and Thomas A. Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 123–138, Austin, Texas, November 1987. ACM SIGOPS.

[BJ87b] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[BJKS88] Kenneth P. Birman, Thomas A. Joseph, Kenneth Kane, and Frank Schmuck. *ISIS — A Distributed Programming Environment User's Guide and Reference Manual.* Department of Computer Science, Cornell University, first edition, March 1988.