Coordinating Access to Computation and Data in Distributed Systems

Douglas L. Thain

A dissertation submitted
in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
(Computer Sciences)

at the
University of Wisconsin - Madison
2004

Abstract

Distributed computing has become a complex ecosystem of protocols and services for managing computation and data. Distributed applications are becoming complex as well. Users, particularly in scientific fields, wish to deploy large numbers of applications with complex dependencies and a large appetite for both computation and data. How may such systems and applications be brought together?

I propose that applications deployed in distributed systems should be represented by an agent. The role of the agent is to transform an application's abstract operations into concrete operations on the varying resources in a distributed system. The agent must hide the unpleasant aspects of individual resources while coordinating their activity in a manner specialized to each application. I examine four open problems in the design of agents for distributed computing.

First, I explore a variety of techniques for coupling a job to an agent, informed by the experience of porting to different systems and deploying with several applications. Coupling an agent to a job via the debugger is by far the most reliable and usable technique and has acceptable overhead for scientific applications.

Second, I describe the problem of coupling an agent to a variety of distributed data systems. This is difficult because of the subtle semantic differences between existing data interfaces. These differences result in the notion of an escaping error, which represents a runtime incompatibility between interfaces.

Third, I present the problem of coupling an agent to a computation manager such as a batch system. This requires a careful discussion of errors in a distributed system. I develop a theory of error propagation and present the notion of error scope, which is needed to guide the propagation of escaping errors.

Finally, I explain how an agent may coordinate the consumption of computation and data resources on behalf of a job. As a case study, I present BAD-FS, a system that executes data intensive batch workloads on faulty distributed systems. I conclude with quantitative evidence underscoring the importance of failure handling in distributed systems.

Acknowledgments

I am thankful to have worked with many fine people throughout graduate school.

Dr. Miron Livny, my thesis advisor, has created a fertile environment of working software, powerful systems, and demanding users that has allowed me and many other students to carry out effective research. Miron has taught me that the hardest and most important job in software design is simply *making it work all the time*. I am grateful for the many opportunities that Miron has provided for me to publish, teach, and travel independently.

Dr. Marvin Solomon has been my mentor in communication. I thank Marvin for taking the time to coach me in effective speaking and writing on nearly every public lecture and academic paper that I have prepared. Marvin also served as a classroom mentor: after taking his class in operating systems, I later taught the same class with his materials and advice.

Dr. Andrea Arpaci-Dusseau and Dr. Remzi Arpaci-Dusseau have encouraged the collaboration of the Condor and Wind projects. I thank both Andrea and Remzi for encouraging both groups to build ambitious systems and target high quality conferences. I thank Remzi for his advice on the early stages of my career. Andrea and Remzi deserve special thanks for providing late night nutrition to all of their students working under deadlines.

Dr. Barton Miller has challenged and encouraged me throughout graduate school. I thank Bart for his instruction, advice, and hard questions in his distributed systems class, in the operating systems seminar, and in my thesis work.

John Bent has been an amiable colleague and invaluable accomplice on many papers and projects in collaboration with the Wind group. I thank John for excellent chalkboard discussions, long working nights, and enjoyable conference trips.

I thank all of the members of the Condor Team for making the third floor an enjoyable and stimulating place to work. In such a large and skilled group, one can always find a technical expert, a project historian, or a new idea. I continue to be impressed by the team's strong work ethic that makes Condor a high quality system.

I am grateful to Cisco Systems, National Cash Register, and Dr. Lawrence Landweber for their support my of education through academic fellowships.

My parents, Gerald and Priscilla Thain, deserve thanks for many intangible gifts, but in this matter, I thank them for encouraging me to be a bookworm.

Last but not least, I thank my lovely wife, Lisa Thain, for providing me with many good reasons to be done with school.

Contents

List of Figures

Chapter 1

Introduction

1.1 Distributed Batch Computing

Computer systems and applications are growing with no apparent bound. As users concoct ever-larger and more complex applications, computer manufacturers respond with larger, faster, and more economical machines, which in turn encourage more aggressive applications. Despite this continuous improvement in capacity, performance, and price, there has always been an important category of users whose needs perpetually exceed the capabilities of any single machine. Such power users may be attempting to solve very large problems by brute force. They may require resources — such as memory — that can only be obtained by aggregating many machines together. They may be engaged in a competition — artificial, commercial, or scientific — where the prize is determined simply by the number of calculations performed.

To achieve such goals, power users require *distributed batch computing systems* that can execute large applications over a long period of time. Such systems tie together hundreds or thousands of machines into one coherent system. Given a structured description of the work to be performed, such a system can distribute work among the available nodes, recover from node crashes and network failures, and generally shield the user from unpleasant events that are the norm in a distributed computing environment. A variety of academic and commercial batch systems are available today.

Historically, batch systems have been geared towards applications that are computation-intensive. An well-known example of a computation-intensive application is the Search for Extraterrestrial Intelligence (SETI) [138], which harnesses cycles from idle personal workstations. The SETI application is a program specially designed to run in the unpredictable environment of a remote workstation. Each available machine is assigned a small data item by a central server, about several hundred kilobytes, computes on that data for a long time, and then returns a result indicating whether that data selection held a signal of interest. The same central server collects all of the results and reports the overall progress of the search back to the project managers. The amortized data needs of the application are less than ten bytes per CPU-second per machine, a rate that easily scales up to a large number of machines without special techniques. Similar comments apply to other computation-intensive applications.

However, there is a growing community of users who hope to apply large scale distributed batch computing to data-intensive applications. Although some such applications may process large *amounts* of data, the distinguishing characteristic is the *complexity* of data interactions. For example, a scientist might wish to run an application drawn from a trusted software repository, seed it with calibration data published by a scientific organization, pass it inputs from a private workstation, run it on a computational cluster managed by an academic department, and then send the results to an archival service. Each component of such a computation might run different software, communicate with different protocols, and be obliged to respect the constraints of different human owners.

As in many computing settings, scientific applications are the first to push the envelope. Data-intensive scientific applications may be found in astrophysics [70], climatology [51], genomics [13], high-energy physics [67], and physical chemistry [139], to name a few. Scientific applications also pave the way for data-intensive commercial applications such as data mining [9], document processing [44], electronic design automation [1], semiconductor simulation [32], financial services [110], and graphics rendering [83].

Computer science has produced countless systems that manage shared computation resources and shared storage independently. There are batch systems, both academic [90] and commercial [65, 157, 5], for managing compute clusters. There are distributed filesystems [126, 68, 78, 8], remote I/O systems [112, 26, 22, 48, 109], and archive managers [130, 93, 23]. A renewed interest in world-wide computational systems known as grids [54] has produced a variety of protocols for accessing both computation [39, 57] and data [11, 149].

Despite the best intentions of their designers, no single system has achieved universal acceptance or deployment. Each carries its own strengths and weakness in performance, manageability, and reliability. A complex ecology of distributed systems is here to stay. The result is that the field of distributed computing is divided into fiefdoms divided by protocol and purpose. One may find systems for managing resources "globally" as long as a user stays within the confines of one organization, one network, or one technology. Such requirements present a problem for data-intensive batch applications that require access to computation and data from many different systems. How are ordinary applications to coordinate their needs for many kinds of resources?

1.2 A Case for Agents

To remedy this situation, I propose that a jobs in a complex system should be represented by an *agent*. The role of the agent is to transform a batch job's abstract operations into concrete operations on the available systems for computation and data. In particular, it hides away the unpleasant aspects of these resources and coordinates their activities in a manner specialized to each application.

This concept is quite similar to how people rely on expert human agents to represent them in specialized matters such as law or real estate. An agent must transform complex, obscure, or specialized language into a form that is understandable to the client. An agent must insulate the client from the vagaries of the field by coalescing multiple messages and hiding temporary setbacks. An agent must coordinate multiple external parties so that they come to agreement on a transaction.

The role of an agent is also strikingly similar to that of a conventional operating system kernel. A kernel is responsible for the many uninteresting bitwise operations necessary to make devices operate as applications expect. It is also responsible for coordinating the use of resources. When an I/O operation causes a delay, a process must be removed from the CPU. When the I/O operation is complete, a process once again regains the CPU. An agent must perform similar tasks in a distributed system.

In this dissertation, I present practical lessons learned from several years of experience building and deploying such agents with working applications and real users in the context of the Condor distributed batch system at the University of Wisconsin. Consequently, I have not focused solely on the traditional computer science problems of performance and efficiency — although those issues deserve some attention — but on the issues of usability, practicality, and reliability that dominate the experience of end users. A significant problem at every level is the question of how to identify, propagate, and react to errors of various kinds. My contribution is an exploration of the *semantics* of agency.

1.3 Overview of Dissertation

This dissertation will explore how data-intensive applications may be run in distributed systems by using agents to mediate the interaction between jobs, data systems, and computation systems. Figure 1.1 depicts the focus of each chapter.

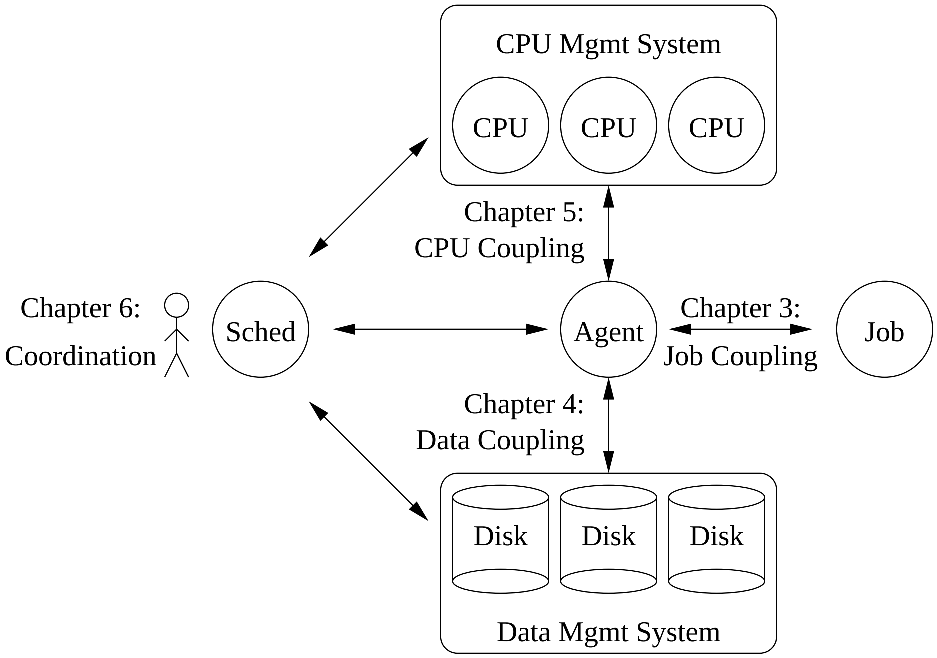


Figure 1.1: Outline of Dissertation

Chapter 2: Background. Distributed computing has historically considered access to computation and access to data as two distinct problems with unrelated solutions. This chapter will review research into previous distributed computing and data systems, paying particular attention to the difficulty of coordinating the two types of systems as they currently stand. It also reviews the literature on interposition agents and the (limited) available advice on error propagation.

Chapter 3: Job Coupling. The vast majority of real programs are not specialized to operate in a distributed system. Most programmers write to the standard interfaces available on workstations, such as the Unix I/O interface. In order to make distributed computing accessible to such conventional applications, we must find ways to seize control of I/O interfaces on standard operating systems without special privileges. This chapter evaluates a variety of techniques for coupling agents to jobs, each with varying degrees of functionality, reliability, and performance. The experience of deployment demonstrates that the primary usability concern is *hole detection*: the ability to determine when the job coupling is incomplete.

Chapter 4: Data Coupling. Once we have established the coupling between an agent and a job, we may turn to the relationship between an agent and the various I/O devices available in a distributed system. Chapter 4 describes Parrot, an agent that can attach a standard Unix application to a variety of I/O protocols used in the scientific community. The mapping of Unix operations to each I/O protocol introduces both semantic and performance problems that must be addressed. This chapter introduces the notion of an *escaping error*, which represents an incompatibility between interfaces.

Chapter 5: Computation Coupling. An agent must also interact with the services provided by a batch system. In order to provide the correct local environment for a job, a batch system must provide services for an agent to interact with at runtime. The chain of services so constructed at runtime is highly susceptible to a wide variety of errors, which can turn into a usability problem. This chapter expands upon the problem of escaping errors, introducing the concept of error scope, and presenting a discipline for error propagation. It demonstrates how this discipline can be applied to agents in both Java and Unix computing environments.

Chapter 6: Coordinating Computation and Data. Chapter 6 ties together the previous chapters by considering jobs, data, and computation as a whole. It introduces the notion of a *workload as a transaction* and discusses the range of ways in which such a transaction may be committed. I will present a case study of BAD-FS: a system that considers both computation and data as first-class resources to be managed using the notion of an entire workload as a transaction. The chapter concludes with a concrete example of the resilience of this approach by running a large workload on a real wide-area distributed system.

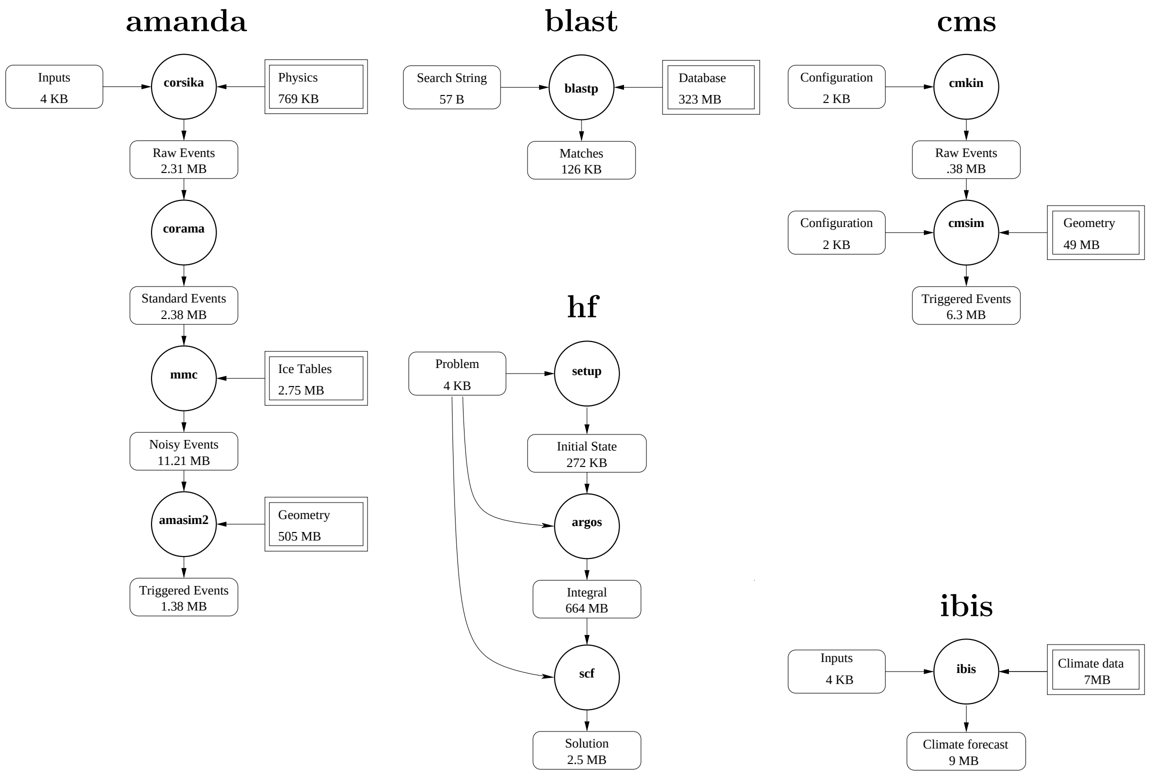


Figure 1.2: Representative Applications

This figure shows the structure of five data intensive applications. Each circle indicates a Unix process. Each box indicates files read or written by the process. Double boxes indicate read-only data shared between many instances of a pipeline.

1.4 Data-Intensive Applications

Throughout this dissertation I will rely on a collection of five data-intensive applications that are candidates for execution in large distributed systems. These applications were drawn from real users at the University of Wisconsin-Madison and represent a range of scientific disciplines. These applications are not a random sampling, but were deliberately chosen because they stress the ability of current systems to deal with the complexity and/or volume of data when run in large numbers. An earlier paper [143] describes these applications in great detail.

Figure 1.2 shows the structure of each of these applications. Each is composed of several standard Unix processes arranged like a pipeline. Each process in the pipeline reads and writes ordinary files in the filesystem. Some files are read-only, while others are written by one process and then read by another. Typically, a large number of pipelines is submitted at once, each with slightly different input parameters, but often sharing the same input files. Although I use the term *pipeline*, these applications do *not* use Unix pipes: they communicate solely through the filesystem.

Each application can solve problems of varying granularity. With guidance from users, I chose problem sizes that were large enough to represent real work, but small enough to remain tractable for analysis and experiment. A single problem can be solved in minutes on a standard workstation, but a real workload would consist of thousands of independent problems.

It is important to note that the schematics shown in Figure 1.2 are given at a level of abstraction comfortable to end users. Invariably, these applications were described in terms such as: *This pipeline expects to find calibration data in* `/data` *and will create a variety of output files in* `/tmp`. In general, users were *not* familiar with low level details, such as the exact subset of files actually used, or whether files were accessed in random or sequential order. This lack of knowledge is a result of the communal nature of scientific applications, which are often built by many developers and may rely on many different libraries of utility code. No single user or developer has a complete low-level picture of how an application operates and are often surprised to discover the underlying details.

AMANDA [70] is an astrophysics experiment designed to observe distant cosmic events. The detector, buried in the Antarctic icepack, observes atmospheric muons produced by neutrinos emitted by the events of interest. The first stage of the calibration software, `corsika`, simulates the production of neutrinos and the primary interaction that creates showers of muons. The second stage, `corama`, translates the output into a standard high-energy physics format. The third, `mmc`, propagates the muons through the earth and ice while introducing noise from atmospheric sources. Finally, `amasim2` simulates the response of the detector to incident muons. The problem size for this application was chosen to be 10,000 initial muon showers.

BLAST [13] searches genomic databases for matching proteins and nucleotides. Both queries and archived data may include errors or gaps, and acceptable match similarity is parameterized. A single executable, `blastp`, reads a query sequence, searches through a shared database, and outputs matches. The problem size for this application was chosen to be a sequence of fifty searches through the standard non-redundant protein database. The fifty search strings were chosen at random from a selection maintained by the University of Wisconsin Biological Magnetic Resonance Bank [6].

CMS [67] is a high-energy physics experiment to begin operation at CERN in 2006. To calibrate the physical detector, it is necessary to simulate its response to the expected collisions. The simulation software is a two-stage pipeline. The first stage, `cmkin`, given a random seed, generates and models particle collisions within the detector. The output is a set of events that are fed to `cmsim`, which simulates the response of the particle detector. The final output represents events that exceed the triggering threshold of the detector. (In fact, CMS will have further stages, but the software for these stages is still in flux.) The problem size for this application was chosen to be 250 collision events.

Messkit Hartree-Fock (HF) [38] is a simulation of the non-relativistic interactions between atomic nuclei and electrons, allowing the computation of properties such as bond strengths and reaction energies. This code is a serialized version of code originally run on parallel architectures. Three distinct executables comprise the calculation: `setup` initializes data files from input parameters, `argos` computes and writes integrals corresponding to the atomic configuration, and `scf` iteratively solves the self-consistent field equations. The problem size was chosen to be the `pwadz` benchmark included with the program.

IBIS [51] is a global-scale simulation of Earth systems. IBIS simulates effects of human activity on the environment, such as global warming. A single program, `ibis` performs the simulation and emits a series of snapshots of the global state. The problem size for this application was chosen to be a one-year simulation at 4 degree resolution.

Much computer science research has concentrated on applications in an interactive workstation environment. For this reason, I will include a sixth application, **Make** as a comparison point. This application is simply a compilation of the standard GNU Bourne Again Shell [3].

1.5 A Note on Terms

Distributed computing uses a variety of often-confused terms to describe the structure of distributed programs. There is no universal agreement upon these terms, but, I will attempt to use them consistently as follows:

- **Program**: A passive sequence of instructions encoded in a programming language.
- **Process**: An active entity running on a machine in a private address space, with one or more threads of control. A process is an operating instance of a program.
- **Job**: A unit of execution in a batch system, consisting of a program to be realized as a process, several specific files to be accessed, and a variety of ancillary details regarding where and how the program is to be run.
- **Workload**: A specific, possibly large, collection of jobs to be run in a batch system, possibly with some partial ordering.
- **Application**: A generic term referring to a general type problem (e.g. weather simulation) that can be attacked by batch computing. This term does not imply any specific computational structure.

Chapter 2

Related Work

2.1 Distributed Batch Systems

The concept of distributed batch computing finds its roots in the earliest commercial computing systems, which operated wholly in batch mode [131]. The cost and complexity of such systems required that multiple users submit their fully-specified programs to a professional operator, along with a description of the resources they intended to consume, using a language such as JCL [31]. The operator would then arrange for queued programs to execute according to some schedule, and return results to each user as programs completed.

The Cambridge Ring [103] can be thought of as an early example of distributed batch computing. The Ring coupled together workstations, servers, and terminals into what we would now call a loosely coupled distributed system. In its time, the Ring was known as a *processor bank* because each processor was considered a generic, substitutable resource. A user would log into a terminal and request any processor of a specific class. The terminal would then connect to that processor and permit the execution of programs. Of course, the Ring was a *interactive* system, but it introduced the notion of generic resources shared among multiple users.

The Condor [90] distributed batch system built upon the idea of the Cambridge Ring by creating a true batch system with respect for the social nature of distributed systems. As in the Cambridge Ring, a Condor user submits jobs that require any processor with particular characteristics. (A specialized language, ClassAds [118], is designed for identifying resources.) Unlike the Cambridge Ring, these jobs are executed in batch mode by a personal scheduler process that identifies and harnesses suitable remote resources for each job. Further, each participant in the system is independent: Each processor is assumed to belong to a primary owner, who is free to evict unwanted jobs at any time. Thus, each scheduler in a Condor system must be prepared for the possibility that a job is evicted and must execute elsewhere. In order to support programs whose runtime may exceed the availability of any one machine, Condor provides a library [133, 91] that allows ordinary programs to migrate from host to host.

A variety of batch systems have followed either the Cambridge or the Condor pattern.

The Cambridge processor bank pattern is found in many of the distributed batch systems marketed in the 1990s and 2000s, such as LSF [157], PBS [65], and SGE [5]. Each of these systems, though physically distributed, is essentially a centralized processor bank owned by a single organization. Each resource in the system is dedicated to the collective good and must accept any work assigned to it. Such systems are generally constructed out of clustered identical machines stored in a professionally managed machine room. It is possible to a build a physically distributed but centrally controlled system: PlanetLab [25] is an example of this. The advantage of this model is that it provides users with fixed schedules and the ability to allocate many machines at once. In these systems, a user may submit, for example, an 8-node parallel job and then be informed that the schedule will allow its execution to begin in 12 hours.

The Condor model has been reflected in several distributed systems, typically specialized to run scientific applications. The Search for Extraterrestrial Intelligence [138] system is the most well known example of this model in the popular press. Others applications include distributed decryption [77] and protein folding [84]. These systems operate in generally the same way: workstation owners are asked to install special software that makes itself known to a centralized project server. When the machine is idle, it requests work from the central server, which is responsible for keeping track of what has been assigned and what is complete.

There also exist a number of interactive distributed operating systems such as Locus [111], Sprite [105], Amoeba [101], Plan9 [107], and MOSIX [21]. These operating systems aim to create an interface very similar to Unix on top of a collection of workstations. Some, such as Amoeba and MOSIX, aim for a *single system image*. That is, regardless of what terminal is used, one sees the same set of processes, files, and so forth. The others allow for varying degrees of location-awareness. In Plan9, for example, one perceives workstations as individual resources, but it is easy to attach a remote filesystem to a local process. Although these systems have made important research contributions in terms of location transparency and recovery from network partitions, they uniformly lack one power key to batch computing: the ability to make a process persistent across crashes. In all of these systems, a newly created process might migrate to other nodes based on sophisticated algorithms regarding data affinity and load balancing. However, if that node should crash, the process will be irretrievably lost.

For this key reason, the model of a Unix-like cluster operating system has not been useful for batch workloads. The complete Unix interface, allowing for interprocess communication, interactive I/O, and so forth, is not easily introduced into a faulty system unless one is willing to mask errors via hardware replication, such as in the Tandem [43] system. A typical batch interface has a simple and idempotent work unit consisting of a single program and its input and output files. The simpler batch interface is more naturally suited to environments where resources may fail.

2.2 Distributed File Systems

Access to data over a network has traditionally been the role of the distributed file system, whose archetypes NFS [126] and AFS [68] have given rise to an enormous number of (largely unsuccessful) variants. Distributed filesystems are almost exclusively focused on the needs of interactive users in centralized administrative environments and local area networks. For example, widely-cited studies of workload patterns have focused on university computer science departments [106, 127, 19, 120] and commercial software development environments [150, 8]. Thus, the performance and semantics of distributed file systems are generally evaluated in terms suitable for these workloads. More evidence of this is that a commonly applied test of distributed filesystem performance is the Andrew benchmark [68], which consists of a series of operations mimicking a program developer. (It is important to note the *local* filesystems are often developed for unique workloads. For example, log-structured file systems [121, 128, 96], are designed and evaluated in the context of write-intensive multimedia and internet applications.)

Consequently, the vast majority of distributed filesystems assume that it is acceptable to expose problems to the user at run-time through ad-hoc interfaces, assuming that there is a human to oversee unusual events. Here are some examples of this orientation.

Typically, NFS is configured in a "soft-mounted" mode so that network disconnections and other errors are exposed to the user after a short time. Thus, any network outage lasting beyond a minute (or whatever time value is configured) will cause all programs using NFS to fail in unusual ways. Such failures are quite acceptable when using an interactive shell to run a single program, but create a frustrating experience for the user that wishes to leave a computer system unattended for days or weeks. The alternative is to configure NFS in a "hard-mounted" mode, which retries errors forever. This mode is also not suitable, because both interactive and batch users may wish to eventually retract or reassign stalled work.

The Coda [78] file system is designed for the interactive user on an occasionally-connected machine such as a personal laptop. When disconnected from a file server, a Coda client attempts to emulate the server as best it can, writing files to cache space and assuming that cache input files are up-to-date. When the client reconnects to the server, it must perform reconciliation by moving newly-written data to the server. If an error occurs during reconciliation — for example, two disconnected clients have written the same file — then the users are informed by email. Again, this channel for communicating errors is acceptable to the interactive user, but would cause chaos in a batch setting.

Buffer servers [15] allow a remotely running program to offload its output to a nearby server and then exit without waiting for all the data to be committed back to the home file server. The buffer server is responsible for this job. However, if the buffer server should fail or be powered off, then the user will silently lose any remotely buffered data. The Unix local buffer cache has essentially the same semantics: A power failure may cause up to thirty seconds of data loss.

These are merely examples of a general orientation to interactive work, and the reader may find many more examples, particularly in the field of peer-to-peer storage [8, 16, 40, 81, 102, 122, 124].

2.3 Data Access in Batch Systems

The Cambridge and the Condor models of distributed batch computing have each been associated with a corresponding mode of data access.

In the Cambridge Ring, access to file data was provided by a distributed file server accessible anywhere from the network. The file service and the compute service were logically distinct entities: one service did not know about the other. Accesses to the file server by programs succeeded or failed solely on their own terms. A failure to read or write data would be exposed to the program and had no direct bearing on access to the CPU. Batch systems in the Cambridge model have preserved this mode of file access. Such systems expect that a conventional distributed filesystem such as is installed by an administrator on all nodes that the batch system is connected to. As in the Cambridge Ring, the success or failure of data access through this system depends entirely on the file system, with no coupling at all to the batch system.

An excellent example of failure independence is found in a recent bug report [2] detailing the interaction between a PBS batch system and an NFS file system. A job submitted to the batch system would write its output to the distributed file system, and then indicate completion to the batch system. A supervising process on another node in the file system would then attempt to read the output file and transmit it to the user. However, the file system was configured to aggressively buffer written in order to maximize performance, so the data were not made immediately available to the supervisor, which concluded that no output had been created by the job. An aberrance in the file system lost the data in transit, yet the job, the file system, the batch system, and the supervisor all concluded that everything was fine.

On the other hand, systems following the Condor model of distributed computing typically have a restricted but highly integrated form of data access. In Condor itself, the batch system is responsible for transferring the data needed by the application between the submission point and the execution site. For jobs that are willing to be re-linked, a remote system call library [133] is provided that allows an application to transparently access data at the submission site, but nowhere else. Both of these functions are tightly integrated with the batch system so that any failure in data transit is immediately attached to the execution of the job, which is then promptly killed and rescheduled elsewhere. However useful and well integrated this method is, the central data service is often a bottleneck.

2.4 Grid Computing

In recent years, the notion of *grid computing* [54, 148] has become popular. At a high level, this idea proposes that computing power and storage capacity should be easily accessible as a professionally managed, inexpensive service, much like electrical and water utilities. The practical application of this idea has been to take resources constructed from local-area distributed systems and add to them an interface appropriate to wide-area distributed systems. Typically, a new interface is required because local-area systems assume a common administrative infrastructure, a relatively trusted user base, and reliable low-latency networks, while grid computing assumes none of these things.

For example, one grid interface to batch computing is GRAM [39]. Typically, an existing batch system is equipped with one submission point exposing a GRAM interface. An external user connects to the GRAM interface and uses GSI [53], a public-key authentication system to authenticate. Once logged in, the user may define and submit batch jobs to GRAM itself. The GRAM interface then turns around and submits the job to the local batch system. In practice, the use of GRAM is not this trivial. Because it is typically used over a wide-area network, failures in the protocol are common, leading to inconsistent state between the user and the GRAM server. When jobs complete in the batch system, they leave behind exit codes and output files that must be communicated back to the user. A tool such as Condor-G [57] may be used to coordinate these activites. A similar service is the UNICORE [12] interface to batch computing. UNICORE is similar to GRAM, but requires the execution site — not the user — to define the set of programs that may be run.

Likewise, grid interfaces have been attached to storage devices, both localized and distributed. What makes grid storage interfaces particular interesting is that they add new semantic capabilities to storage, even though all are built on top of ordinary local filesystems. For example, the GridFTP [11] storage server adds public key cryptography to the venerable FTP [112] protocol. DCache [48] is a hierarchical storage manager designed to optimize the use of disk cache space for workloads that move large data volumes between tapes and running applications. IBP [109] allows for time-limited allocation of extent-structure storage with a security system based on capabilities [86]. NeST [26] also allows for time-limited allocation of space, but provides Unix-structured storage along with AFS-like access control lists. RFIO [22] is designed for small-granularity access to a mass storage archive. SRB [23] is designed to provide access to highly-structured data spanning both filesystems and databases. SRM [130] also provides time-limited allocation, but provides a semantic framework distinguishing temporary, durable, and persistent storage.

What makes these grid storage devices different from traditional distributed filesystems is that each device is an independent entity. There are no overarching consistency semantics. There is no uniform naming scheme. There are no transaction semantics for adding, removing, or deleting data from multiple services. Clients of grid storage devices must build up these higher level properties on top of individual storage devices.

2.5 Interposition Agents

The term *interposition agent* was coined by Michael Jones [75] to refer to code transparently placed between a process and the operating system. Jones relied on the Mach [7] feature that allows system calls to be reflected back into the address space of the process that issued them. Jones' work is unique because the agent resides inside the process, yet traps operations based on the external interface to the operating system.

Several techniques for building interposition agents have been devised, each with various strengths and weaknesses.

Many techniques rely on the structure of library interfaces. The Condor remote system call library [133] is linked to an application in the ordinary way. More transparent tools include Detours [71] and Mediating Connectors [20] which attach via the mechanism of dynamic linking in Windows operating systems. Another example is the Knit [119] toolkit, which encourages the construction of code modules with compatible interfaces that may be assembled in a variety of configurations at build-time.

The technique of interposition via the debugger interface, which will be explored in detail below, is proposed by the UFO [10] system. The debugger is also employed by techniques related to interposition, such as sandboxing [61] and virtual operating systems [42].

There are a variety of interposition techniques that require privileged access to the operating system. For example, the SLIC [59] and FIST [154] toolkits encourage the construction of stackable device drivers and file systems that can be reconfigured according to local needs. Small surgical changes may also be made to an operating system in order to allow a user process to serve as a kernel extension. Such a facility may be present from the ground up in a microkernel such as Mach [7], but can also be added as an afterthought, which is the case for most implementations of AFS [68]. In the early 1990s, such techniques were generalized under the heading of "extensible operating systems" [27, 47, 129], however such features have not entered the mainstream.

The experimental operating system Plan 9 [108] provides extensibility through the normal filesystem interface. Applications are permitted to construct private namespaces and may create new filesystems and devices on the fly by designating user-level programs that implement the needed functionality.

Another popular technique is to interpose by substituting a standard file server with an enhanced service. The NFS protocol is widely used for this purpose. For example, the Alex [33] service provides an FTP cache behind a standard NFS service. Similar ideas are found in the Legion [152] object-space translator and the Slice [14] routing microproxy.

Services offered through NFS interposition typically have severe semantic constraints due to the nature of the protocol itself. For example, the stateless NFS protocol has no representation of the system calls `open` and `close`, so one cannot tell when files are actually in use. Further, such file system interfaces do not express any binding between individual operations and the processes that initiate them. That is, a remote filesystem agent sees a `read` or `write` but not the process ID that issued it. Without this information, it is difficult or impossible to perform accounting at the server side for the purposes of security or performance. (Reumann and Shin [123] have proposed techniques for carrying such context through the many layers of a system.)

Despite all of this related work describing the feasibility of interpositioning using various techniques, there has been little discussion about the practicality and effectiveness of these techniques on real applications. My contribution in Chapter 3 will be to describe the effectiveness of these techniques on real applications in real systems.

2.6 Error Handling

Throughout this dissertation, a running theme will be the problem of error handling in various contexts. Although errors must be a concern in a software project of any scope, there is very little work that offers advice on how to structure and propagate errors beyond the most vague generalizations.

For example, my work makes use of multiplexed interfaces: Multiple I/O services are hidden behind a uniform library interface. One can easily find multiplexing in a variety of settings, including the Sun Virtual Filesystem Switch (VFS) [79], the user-level Uniform I/O Interface (UIO) [36], the NeST [26] storage appliance, the GRAM [39] interface, the Proteus [37] message library, and the Ace [117] language, to name a few. Despite the ubiquity of this technique, I am not aware of any detailed treatment regarding the problem of failures and interface mismatches when multiplexing existing interfaces. The closest such discussion is a report by Craig Metz [97] on the correct use of the multiplexed Berkeley sockets interface.

Chapter 4 will address the problem of multiplexing in detail.

Likewise, all of the previous work on interposition has described the procedure only in the most positive terms. The reality is quite the opposite, and I will expand upon upon it in Chapter 3. I am aware of one exception that describes the problems of interposition: T. Garfinkel [58] describes how the techniques of sandboxing are subject to an enormous number of subtle bugs.

The notion of *escaping error* introduced in Chapter 5 finds its genesis in the discipline of *design by contract*, proposed abstractly by Hoare [66] and developed more concretely by Meyer[98]. Similar hints are also given by Goodenough [62], Ekanadham and Bernsteien [45], and Howell and Mularz [69]. In all these works, the escaping error is usually implemented by an instruction that brings the entire computation to a halt with a message to the console. A global halt is neither possible nor desirable in a distributed system.

A common refrain is that errors are easily handled as long as the programmer is diligent in using *exceptions*. Exceptions as a language feature is generally attributed to Goodenough [62], and has progressed through a variety of languages from research to commercial use, including CLU [88], Haskell [95] Ada [69], C++ [46], and Java [17]. However, the exception has not received universal approbation. Black [30], Chen [35], and Spolsky [136], among others, have made credible arguments against the use of exceptions. The many variations on the exception concept have disagreed on whether an interface must declare all possible exceptions present in the implementation. Chapter 5 will address the problem of exceptions in detail.

Chapter 3

Job Coupling

3.1 Introduction

In the previous chapter, I reviewed a wide variety of systems available to users for managing distributed computation and data. How can we connect ordinary programs to these systems without placing any burden upon the user, such as re-coding, re-compiling, or re-linking?

The reader may think this perspective to be a little unusual. After all, one may easily access remote storage from a personal workstation by simply installing a remote filesystem client and server on the necessary machines. However, consider the typical consumer of a batch computing system. Using a few simple commands, the user may be able to gain control of hundreds or thousands of CPUs spread across an entire university or business. Typically, those CPUs will be under the control of remote workstation owners or system administrators, so running any sort of privileged program, much less making changes to the kernel, is not likely to be permitted.

Moreover, a user is likely to submit ordinary programs that were created and tested on ordinary workstations, using ordinary interfaces to perform ordinary input and output. Such users often wrap programs in scripts in order to invoke standard system utilities to move files, compress or filter data, and generally set up the appropriate environment for the core program. Although *some* users might be willing to re-code *some* programs, it is unlikely

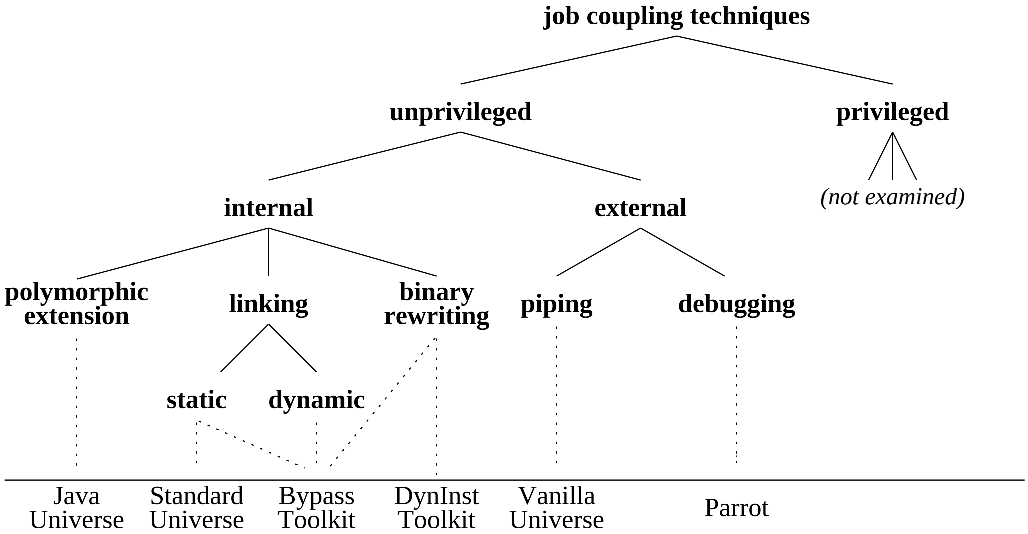


Figure 3.1: Taxonomy of Job Coupling Techniques

that they would be willing to do the same to the entire library of system tools in common use.

To make distributed computing accessible to standard applications, we require the ability to slip a new piece of code between an application and the operating system. Such code can trap the external operations of a program and direct them to other services, leaving the program none the wiser. Michael Jones [75] coined the term *interposition agent* (or simply *agent*) to describe such code.

There are many techniques for coupling an agent to a program. Each has particular strengths and weaknesses. Although some of these techniques have been described by other researchers, there has previously been little evaluation of their relative merits beyond a simple comparison of performance. My contributions in this area include the practical lessons learned by constructing and deploying each of these techniques in production computing settings with the Condor distributed batch systems.

Figure 3.1 shows a taxonomy of job coupling techniques. (Notations at the bottom of the figure indicate the toolkit or environment that I used or built to evaluate each technique.)

At the highest level, they are divided into two groups: those that require privileged access to the operating system, and those that do not. For the reasons stated above, I will not present any techniques that require special privileges, although some have been reviewed in Chapter 2. Unprivileged techniques may be further divided into internal and external. Internal techniques modify the memory space of the program in some fashion. In general, these techniques are highly flexible and impose little performance penalty, but are difficult to port and develop, and cannot be applied to arbitrary programs. The internal techniques include polymorphic extension, static and dynamic linking, and binary rewriting. External techniques capture and modify operations that are visible outside an application's address space. In general, these techniques are less flexible and incur a performance penalty, but are highly reliable and can be applied to most programs.

3.2 Internal Techniques

3.2.1 Polymorphic Extension

The simplest internal technique is *polymorphic extension*. If the internal code of the application is amenable to extension, we may simply add a new implementation of an existing interface or class. The user then must make small code changes to invoke the appropriate constructor or factory in order to produce the new object. This technique has been deployed in two contexts in the Condor distributed batch system.

The Java Universe [146] within Condor provides the user with two objects, `ChirpInputStream` and `ChirpOutputStream`, that match standard stream interfaces for I/O. The application programmer then replaces objects of type `InputStream` and `OutputStream` with their analogues in the source code. The new objects types adapt ordinary stream I/O into the Chirp service, which is described in Chapter 4.

The Java language has facilities for *reflection*, which allows external classes to be loaded and examined within a program itself. A program structured to use reflection is well-suited for interposition without any code changes as all. The difference boils down to the distinction between constructors, which hard-code the class of an object, and factories, which delegate the class name to be loaded.

The ROOT I/O library [4] is a multipurpose C++ library used in the high-energy physics community. ROOT multiplexes a variety of I/O protocols into one interface, presenting them under a unified name space with a URL like syntax. A Chirp object in ROOT allows applications linked with the changed ROOT library to access Chirp services by merely accessing a different logical file name.

Strictly speaking, this coupling technique might not qualify as interposition because it requires changes, albeit small, to the source code of a program. Accordingly, we will consider it as a desirable boundary case in comparison with the other techniques. Polymorphic extension has no further complexities in job coupling, but is still subject to the problems of I/O and CPU coupling described in Chapters 4 and 5.

3.2.2 Linking

An agent may be bound to a program using the standard system linker, either at build time or by the dynamic linker at run time. This technique is initially very attractive, because trivial agents have a trivial implementation. For more realistic agents, this technique has many complexities that make such agents very difficult to use and expensive to maintain.

Let us begin with an attractively simple example. Suppose that we wish to create an agent that traps all of an application's `open` operations. A simple agent might record a log message for each `open` and then allow the operation to continue unchanged. Typically, `open` appears in the standard system library as an ordinary routine that in turn invokes an operating system call to actually open the file. So, to gain control of all `opens`, we create a fragment of code that replaces the existing definition:

```
int open( const char *path, int flags, int mode ) {
        write_log_message("aha! you opened %s",path);
        return syscall(SYS_open,path,flags,mode);
}
```

Once compiled, this fragment can be statically inserted into a program at build time by adjusting the linking instructions, or it may be forced into any dynamically-linked program at run time by issuing special instructions to the run-time linker. Many variants of Unix allow an environment variable such as LD_PRELOAD or _RLD_LIST to indicate the name of the agent library.

Although the two forms of linking are technically very similar, the dynamic linker has several advantages. The static linking technique must be applied on a program-by-program basis at build time, thus requiring the user to have some degree of technical sophistication and access to the object code of the application. The dynamic linking technique can be applied by non-programmers at run-time to a large number of programs; typically, all of the standard tools found on a modern Unix system are dynamically linked. Further, the dynamic technique may be applied to operations that are not necessarily system calls, because the dynamic linker provides routines for explicitly discovering and invoking function calls in other libraries.

For example, the `fopen` operation is typically not a system call, but rather a C library function. The dynamic linker could be used to trap and invoke `fopen` as follows:

```
FILE * fopen( const char *path, const char *flags ) {
        write_log_message("aha! you opened %s",path);
        library_pointer = dlopen("libc.so",RTLD_LAZY);
        function_pointer = dlsym(library_pointer,"fopen");
        return (*function_pointer)(path,flags);
}
```

(Of course, in a real agent, the fragment above would also check for errors, as well as cache the results of `dlopen` and `dlsym`.) Regardless of the linking method, these sorts of simple agents can work on trivial programs, but have some complexities that must be addressed in order to make them work on non-trivial programs. Some of these complexities are intrinsic problems that arise from the essential nature of such agents. Others are practical complexities that arise from the realities of the surrounding system.

Intrinsic Complexities

Most agents intend to invoke complex code whenever they gain control of an operation. Such code is almost certain to invoke the originally trapped operation. Suppose that `write_log_message` must open a file in which to place its log messages. If it invokes `open` directly, it will accidentally invoke the definition of `open` in the agent, recurse, and crash.

One could avoid this problem by simply being very careful never to invoke trapped functions within an agent. This restriction can be an enormous burden on development, generally requiring all code in the agent to be hand-built without assistance from outside libraries, which are almost certain to invoke standard system calls. Although development under this restriction is possible, my experience with this technique is quite negative. For example, the Condor checkpointing library [133] maintains this attempt at purity, and is thus a very difficult piece of code to maintain, typically being the last and most expensive component ported to new operating systems.

A better solution is to introduce the notion of *layers*. The entire assembly of the program, the agent, and the standard library may be thought of as a stack of software layers. A running program has one *active layer* in which it currently executes. A program begins in the topmost layer. If it invokes a trapped function, the active layer is adjusted to point to the agent. Further calls to the same function are directed into the bottommost layer, which is the standard library. This fragment demonstrates the layering technique:

```
typedef enum {PROGRAM,AGENT,STDLIB} layer_t;
layer_t active_layer = PROGRAM;

int open( const char *path, int flags, int mode ) {
        int result;
        if(active_layer==PROGRAM) {
                active_layer = AGENT;
                write_log_message("aha! you opened %s",path);
                result = open(path,flags,mode);
                active_layer = PROGRAM;
        } else {
                active_layer = STDLIB;
                result = syscall(SYS_open,path,flags,mode);
                active_layer = AGENT;
        }
        return result;
}
```

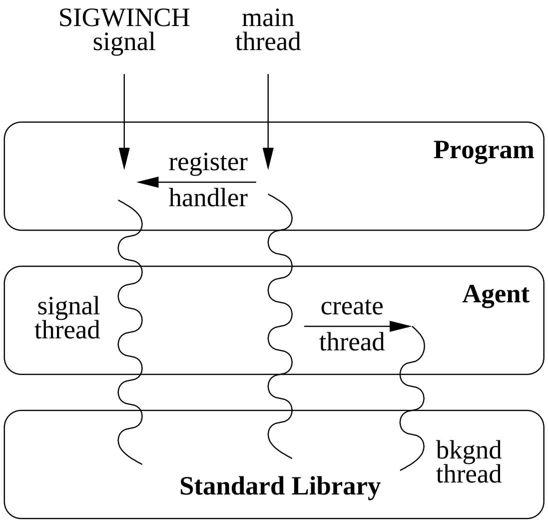


Figure 3.2: Threads and Layers

With this technique, the agent may invoke arbitrary code, including ordinary calls to the trapped function. Interior invocations of `open` will detect the change in layer and be routed to the original system call or library routine. Naturally, one may have an arbitrary number of layers rather than the fixed three shown here.

The active layer record may be thought of as an annotation to the stack of activation records in the flow of control. A function invoked recursively within an agent is logically a different function within the same program. It follows that a program with multiple stacks must have one active layer record for each stack. A program that uses multiple threads or makes use of signal handlers has a separate stack for each thread or handler, and thus must associate an active layer with each.

To implement the active layer record for such programs, the agent must not only trap calls to functions of interest, but also thread creation and deletion, and the installation, execution, and removal of signal handlers, so that each may have its own active layer record. A newly-created thread or signal handler is intended to execute within the semantic context of the code that created it. Thus, each is given the active layer of the parent that created or installed it.

Figure 3.2 shows an example of this concept. Suppose that a program is run with an agent attached. The program wishes to be notifed when its window size changes, so it establishes a signal handler for **SIGWINCH**, The agent traps this attempt to establish a signal handler and records that the caller's active layer was **PROGRAM** when the handler was established. Now, suppose that the program invokes a function such as `open` that is trapped by the agent. The main thread's active layer is changed to **AGENT** in the process. The agent desires to perform some garbage collection in the background, so it creates a thread to do so. The new thread is given an initial active layer of **AGENT**, the same as its parent thread. The main thread completes its business in the agent and returns to the program, while the background thread continues in the agent. If a **SIGWINCH** signal arrives, the signal handler will be executed in the **PROGRAM** layer, regardless of the thread that is currently active. Of course, if that signal handler should invoke a function trapped by the agent, it will be free to descend to lower layers just like any other thread.

Naturally, in any program where either the agent or the program are multi-threaded, the usual care must be taken when accessing shared resources such as the main memory allocator. The agent framework cannot relieve the programmer of this responsibility.

Practical Complexities

Beyond the problems already described, there are a number of practical complexities involved in linking that make development a very labor-intensive activity. From the experience of porting the dynamic linking technique to a variety of Unix-like systems, I have identified the following classes of problems:

Multiple entry points. Most Unix variants have a number of variations on each system call within the standard library. For example, `open`, `_open` and `__open` can be aliases for a single underlying function `__libc_open` that actually invokes the underlying system call. The number and names of these entry points varies from system to system.

Obscured interfaces. The `stat` system call returns summary information about a file. The structure returned by `stat` has changed as architectures have moved from 16 to 32 to 64 bits. As a result, the `stat` defined in most standard libraries assumes an obsolete definition of the structure. Recent programs that appear to use `stat` at the source level are actually redirected, by way of a macro or inline function, to a system call often named `fxstat`. However, for backwards compatibility, the standard library (and a complete agent) must provide the old `stat` and `stat64` as well.

Varied implementations. `socket` is a well-known library interface for creating a communication channel. However, several systems do not implement `socket` by invoking a matching `socket` system call. Some systems implement it as `open` on a special file, followed by an `ioctl`. Others implement it as a call to `so_socket`, whose additional arguments and semantics are undocumented.

The Bypass Toolkit

For all of these reasons, building even simple library-based agents is a complicated matter, requiring not just knowledge of the *interface* to the library, but also fair knowledge of the *implementation* of the library. Some of this complexity can be encapsulated within a helper library, but much of it appears as cross-cutting code in many entry points to the agents.

To assist with the construction of such agents, I have created the Bypass toolkit [144, 145]. This toolkit accepts a high-level description of an agent from a programmer, omitting all of the aforementioned ugly details:

```
int open( const char *path, int flags, int mode )  {{
        write_log_message("aha! you opened %s",path);
        return open(path,flags,mode);
}}
```

The Bypass code generator reads this pseudo-code, and then generates source code for an agent, taking into account all of the intrinsic problems of layering, threading, and signal safety. In addition, it consults a *knowledge file* that accounts for all of the practical complexities that differ from platform to platform. Finally, the generator emits source code that can be linked either statically or dynamically with a user's program.

It should be noted that the construction of an accurate knowledge file for a given platform is a non-trivial task, requiring some knowledge of the workings of the standard library on that platform. However, it can be done, and Bypass includes a reasonably complete knowledge file for several versions of Linux, Solaris, IRIX, HP-UX, and OSF/1.

3.2.3 Binary Rewriting

Another technique for coupling a job to an agent is *binary rewriting.* This technique involves making surgical machine code changes to a running program in order to redirect the control flow from one point to another. This technique requires detailed knowledge of both an instruction set and the common assembly idioms emitted by a compiler. These problems have been solved by the DynInst [99] toolkit, which encapsulates the details in a architecture-dependent library that allows the rewriting of arbitrary code at any point within a program.

Binary rewriting has been been demonstrated as a viable job coupling technique by Victor Zandy in both the Condor system call libraries [156] and the Rocks [155] toolkit. To demonstrate the generality of this technique, I have added a binary rewriting capability to Bypass, thus allowing general agents to be coupled to programs in this manner.

The advantage of using the binary rewriting technique is that it can be used to couple to any function, whether dynamically or statically linked. In the latter case, the executable must have debugging data in the executable in order to identify the address of the function entry. The disadvantage is that the binary rewriting is processor dependent, whereas the library technique is processor independent. Unchanged is the requirement that the agent programmer (or Bypass) must still know all of the gritty library details and construct compensating code.

3.3 External Techniques

The difficulties of the internal techniques stem from the need to understand the detailed structure of a program and the libraries that it interfaces with. Many of these problems can be avoided by placing the agent in an external process and coupling it to the program through public interfaces in the operating system kernel.

3.3.1 Streaming

A simple external technique is to make use of the existing stream interfaces in Unix. A program's standard I/O streams may be connected to an agent on the same host via Unix pipes or an agent on another host via TCP streams. An event driven agent can watch for new data on output streams and available space on input streams, pulling and pushing data as necessary. Stream interfaces are heavily used and thus highly optimized in most operating systems, allowing data to pass between program and agent bound only by memory (or network) bandwidth.

Of course, as a job coupling technique, streaming is only useful to programs that perform only streaming I/O. If the program requires access to files through the conventional open/read/write/close interface, then another technique must be chosen.

The only wrinkle in this technique is that the agent must consider the exit status of the program in order to understand the final disposition of the I/O streams. The agent will always see an end-of-file condition on each stream, regardless of whether the program exits normally, crashes, is killed by another user, or even fails to begin execution. The mere acceptance of available output on a stream cannot be considered evidence of a successful execution.

3.3.2 Debugging

Operating systems provide a specialized interface for a debugging tool to stop, examine, and resume arbitrary programs. The debugging interface can also be used as a job coupling technique. Instead of merely examining the debugged program, the debugging agent traps each system call, provides a new implementation, and then places the result back in the target process while nullifying the original system call.

Note that this technique interposes on the system call interrupt issued by the application. It does not manipulate any of the library routines associated with a system call. By relying on the interrupt mechanism, this technique ensures that the agent is capturing all of the application's system calls. The importance of this completeness will be emphasized below. The debugger could be used to manipulate library calls, but would suffer many of the same complexities of the internal techniques described above.

Alexandrov et al. [10] have described the use of the Solaris `proc` debugging interface to instrument a process in this manner. However, Linux is currently a much more widely deployed platform for scientific and distributed computing. Its `ptrace` debugger model is generally considered inferior to the Solaris `proc` model; it can still be used for interposition, but it has limitations that must be accommodated. (Moreover, Alexandrov did not address the semantic problems of agency that I will expand upon below.)

Figure 3.3(a) shows the control flow necessary to trap a system call through the `ptrace` interface. The agent process registers its interest in an application process with the host kernel. At each attempt by the application to invoke a system call, the host kernel notifies the agent of the attempt. The agent may then modify the application's address space or

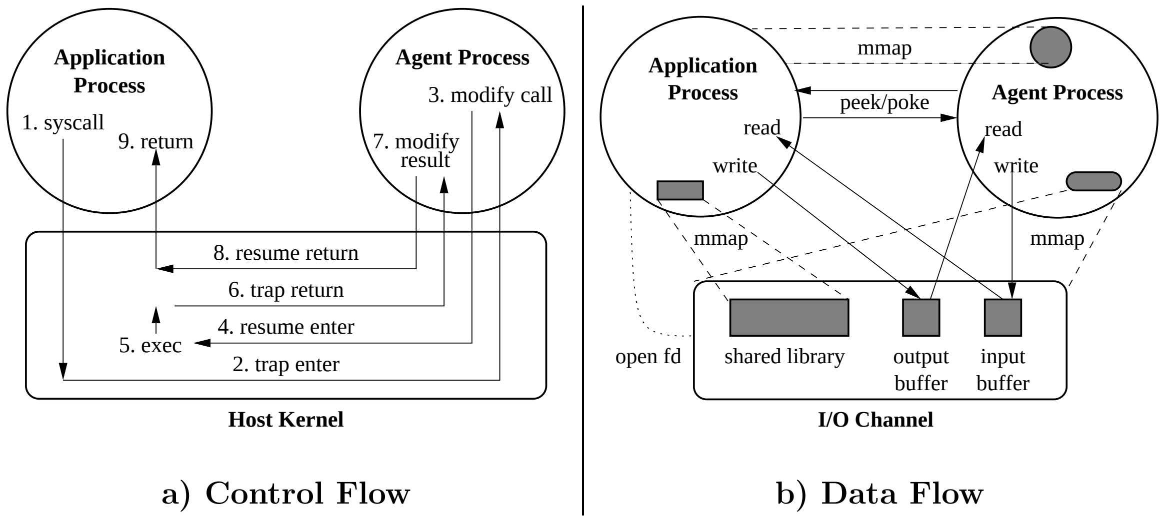


Figure 3.3: Job Coupling via the Debugger

registers, including the system call and its arguments. (If a system call is to be entirely replaced by the agent, it may simply redirect the physical system call to `getpid`, which is fast and inexpensive.) Once satisfied, the agent instructs the host kernel to resume the system call. At completion, the agent is given another opportunity to modify the application and the result. Once satisfied, the agent resumes the return from the system call, and the application regains control.

There are two complexities with this approach:

Process ancestry. The `ptrace` interface forces all traced processes to become the immediate children of the tracing processes. A change in ancestry is needed because notification of trace events occurs through the same path as notification of child completion events: the tracing process receives a signal, and then must call `waitpid` to retrieve the details. As a consequence, any tracing tool that wishes to follow a tree of processes must maintain a table of process ancestry. All system calls that communicate information about children (such as `waitpid`) must be trapped and emulated by the agent. If a traced process forks, the Linux kernel (inexplicably) does not propagate the tracing flags to the child. This omission may be overcome by trapping instances of `fork` and converting them into the more flexible (and Linux specific) `clone` system call which can be instructed to create a new process with tracing activated.

Data flow. The emulation of system calls requires the ability to move data in and out of the target application. Figure 3.3(b) shows all of the necessary data flow techniques. The most convenient would be to access a special file (`/proc/n/mem`) that represents the entire memory space of the application. This file can be modified with standard I/O operations, and can also be mapped into the address space of the agent process.

Although the memory-mapped file provides high-bandwidth read access, writing to this file is not permitted by the kernel. Although the kernel developers have attempted to add support for writing to this file, it never achieved stability and it introduced a security hole whereby a debugger could retain write access to a child that had elevated its privilege level. Two considerations indicate that this functionality is not likely to return. First, Linus Torvalds, the author of the kernel, has expressed disapproval of the form of the interface. Second, a correct implementation would be quite complex because it requires two levels of memory mapping: one to express the memory layout of the debugger, and another to express the memory layout of the debuggee.

However, a simpler interface can be relied upon. A pair of `ptrace` calls, `peek` and `poke`, are provided to read or write a single word in the target application. This interface can be used for moving small amounts of data into the target application, but is obviously not suited for moving large amounts of data such as is required by the `read` and `write` system calls.

To move data efficiently, the application must be coerced into assisting the agent. Coercion can be accomplished by converting many system calls into `preads` and `pwrites` on a shared buffer called the *I/O channel*. The I/O channel is an ordinary file, created by the agent, passed implicitly, and shared among all of its children. (Recall from above that all descendants of the agent are forced to become its immediate children.) The agent maps the

	internal techniques				external techniques	
	polymorphic extension	**static linking**	**dynamic linking**	**binary rewriting**	**stream**	**debug**
applicability	one lib	one pgm	dyn libs	dyn prog	any	not setuid
burden	change code	relink code	identify	identify	run cmd	run cmd
flexibility	fixed	any	any	any	stream	syscall
init/fini	hard	hard	hard	hard	easy	easy
debugging	joined	joined	joined	joined	separate	separate
security	no	no	no	no	yes	yes
hole detection	easy	hard	hard	hard	easy	easy
porting	none	by os	by os	by os/cpu	none	by os

Figure 3.4: Comparison of Job Coupling Techniques

I/O channel into memory, in order to minimize copying, while all of the application processes simply maintain a file descriptor pointing to the I/O channel.

For example, suppose that the application issues a `read` on a remote file. Upon trapping the system call entry, the agent examines the parameters of `read`, retrieves the needed data, and copies it directly into a buffer in the I/O channel. The `read` is then modified (via `poke`) to be a `pread` that accesses the I/O channel instead. The system call is resumed, and the application pulls in the data from the I/O channel, unaware of the activity necessary to place it there.

3.4 Usability

So far, I have outlined the technical challenges necessary to build each of these job coupling techniques. These technical issues have a direct impact on the usability of each technique, summarized in Figure 3.4.

Applicability. The four internal techniques may only be applied to certain kinds of programs. Polymorphic extension and static linking only apply to those programs that can be rebuilt. The dynamic library technique requires that the library to be replaced be dynamically linked, while binary rewriting simply requires the presence of the dynamic loader. Both of the external techniques can be applied to any process at all, regardless of how it is linked, with the exception that debugging prevents the program from elevating its privilege level via the `setuid` feature. The debugger technique, in particular, easily works with entire trees of processes, and is thus naturally suited for use with scripting languages.

Burden. Each technique imposes a different burden on the end user to couple a program and agent together. For example, polymorphic extension requires small code changes while static linking requires rebuilding. These techniques are generally acceptable for custom-built code, but are usually not possible with packaged commercial software. Dynamic linking and binary rewriting require that the user understand which programs are dynamically linked and which are not. Most standard system utilities are dynamic, but many commercial packages are static. My experience is that these techniques *appear* to have a low burden – the user simply sets an environment variable – but users are surprised and quite frustrated when an (unexpectedly) static application blithely ignores an interposition agent. Both streaming and debugging require the user to explicitly invoke a command in conjunction with a program. However, the debugger technique can be invoked once to open a shell, rendering further use transparent because it operates on all descendants of the shell.

Flexibility. Perhaps the most significant difference between the techniques is the varying degree of flexibility in trapping different layers of software. Naturally, polymorphic extension is fixed to interfaces specified by the software framework. However, the other internal techniques can be applied to gain control of any layer of code. For example, Bypass has been used to instrument an application's calls to the standard memory allocator, the X Window System library, and the OpenGL library. The external techniques of streaming and debugging are fixed to the particular interfaces of streams and system calls, respectively.

Initialization and finalization. Many important activities take place during the initialization and finalization of a process: dynamic libraries are loaded; constructors, destructors, and other automatic routines are run; I/O streams are created or flushed. During these transitions, the libraries and other resources in use by a process are in a state of flux and may not be usable by the agent. This transitory state complicates the implementation of internal agents that wish to intercept such activity. For example, the application may perform I/O in a global constructor or destructor. Thus, an internal agent itself cannot rely on global constructors or destructors: there is no ordering enforced between those of the application and those of the agent. The programmer of such agents must exercise care not only in constructing the agent, but also in selecting the libraries invoked by the agent. These activities are much more easily manipulated through external techniques. For example, the debugging technique can be used to trap and modify the behavior of the dynamic linker, perhaps to allow dynamic libraries to be loaded from a remote source. Such actions are not possible using an internal agent.

Debugging. No code is ever complete nor fully debugged. Production deployment of interposition agents requires the ability to debug both applications and agents. All of the techniques admit debugging in some form or another, but all complicate the matter in some way. Because internal agents are joined to the address space of their targets, a single debugger may be used to trace the control flow from program to agent. However, this convenience also joins the fate of the two programs: it is difficult to determine whether a crash is the fault of the agent or of the program. When using streaming or debugging, both the agent and the program must be debugged separately, but one's failure is isolated from the other. The debugging technique is further restricted by the fact that only one process may attach to the target program at once. However, a debugging agent may be used to invoke an entire process tree, so that the target program may be debugged so long as the (non-agent) debugger is also a child of the (agent) debugger.

Security. Interposition agents may be used for security as well as convenience. An agent may provide a *sandbox* [61, 58] which prevents an untrusted application from modifying any external data that it is not permitted to access. The internal techniques are not suitable for enforcing security because they may easily be subverted by a program that invokes system calls directly without passing through libraries. The external techniques, however, cannot be fooled in this way and are suitable for enforcing security.

Hole detection. Closely related to security and debugging is the matter of *hole detection*. Inevitably, an agent will fail to trap an operation attempted by an application. (To wit, the agent has a *hole*.) The hole may simply be a bug in the agent, or it may be that the interface has evolved over time, and the application is using a deprecated or newly added feature of which the agent is not aware. Internal agents are frightfully sensitive to holes. As standard libraries develop, interfaces are added and deleted, and modified library routines may invoke system calls directly without passing through the corresponding public interface function. Such silent changes cause general chaos in both the application and agent, often resulting in crashes or (worse) silent output errors. No such problem occurs in external agents, particularly when using the debugger. Although the system call interface can change, an unexpected event at this layer is an explicit event that the agent can trap. It may then terminate the application and indicate the exact problem.

The problem of hole detection must not be underestimated. My experience with porting Bypass to five operating systems and many minor system versions is that any significant operating system upgrade includes changes to the standard libraries, which in turn require modifications to internal trapping techniques. Thus, internal agents are *never* forward compatible across minor operating system upgrades. Further, identifying and fixing such holes is time consuming. Because the missed operation itself is unknown, one must spend long hours with a debugger to see where the expected course of the application differs from the actual behavior. Once discovered, a new entry point must be added to the agent. The treatment is simple but the diagnosis is difficult.

Porting. For these reasons, I have described *porting* in Figure 3.4 as follows. The polymorphic extension requires no porting effort, as it depends on the application framework

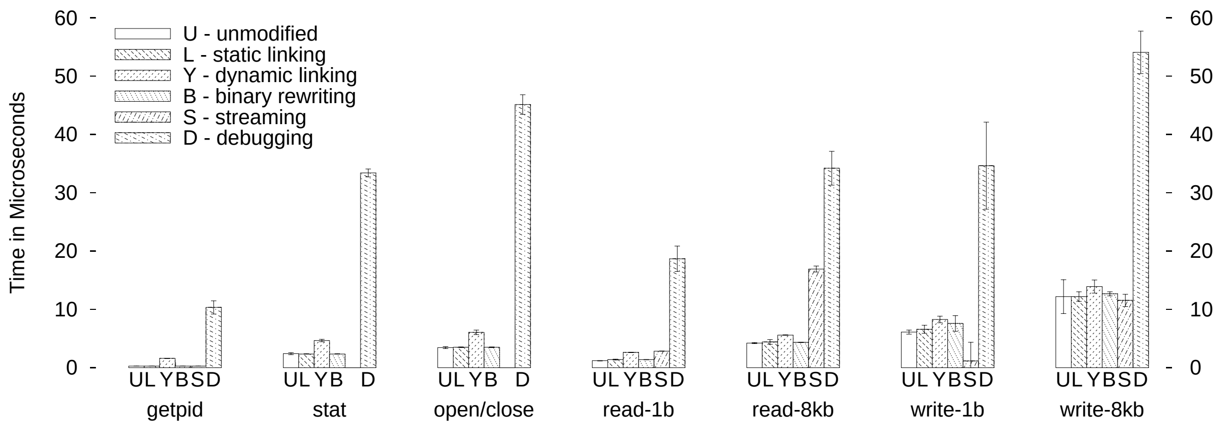


Figure 3.5: Latency of Coupling Techniques - Graph
This figure graphically compares the overhead of each coupling technique. For each system call, a bar shows the latency and variance of each technique. Note that the stream interface does not include the `stat` *or* `open/close` *calls. Detailed figures may be found in Figure 3.6.*

rather than the running platform. The debugger has significant operating system dependent components that must be ported, however the nature and stability of this interface makes for a tractable task. The remaining techniques – static linking, dynamic linking, and binary rewriting – should be viewed as a significant porting challenge that must be revisited at every minor operating system upgrade.

3.5 Performance

Figure 3.5 graphically compares the latency overhead of all of the job coupling techniques. Figure 3.6 shows the same data in tabular form. (I have omitted the polymorphic extension, as the cost of that technique is determined by the application framework.) Each entry was measured by a benchmark C program which timed 1000 cycles of 100,000 iterations of various system calls on a 1545 MHz Athlon XP1800 running Linux 2.4.20. Each system call was performed on an existing file in an ext3 filesystem with the file wholly in the system buffer cache.

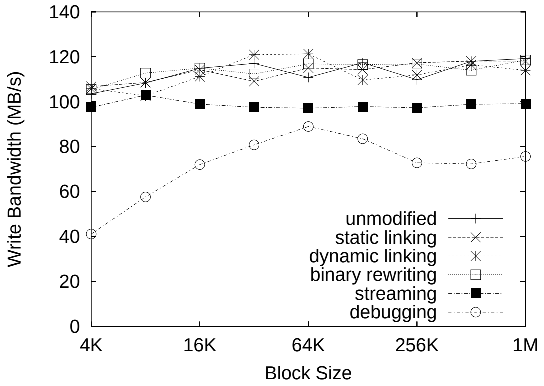
	getpid		stat		open/close		read-1b		read-8kb		write-1b		write-8kb	
	μs	α	μs	α	μs	α	μs	α	μs	α	μs	α	μs	α
unmod	0.28	*1.00*	2.40	*1.00*	3.45	*1.00*	1.20	*1.00*	4.22	*1.00*	6.11	*1.00*	12.19	*1.00*
static	0.28	*1.01*	2.36	*0.98*	3.52	*1.02*	1.40	*1.16*	4.42	*1.05*	6.58	*1.08*	12.19	*1.00*
dynamic	1.61	*5.79*	4.65	*1.94*	6.06	*1.76*	2.62	*2.18*	5.59	*1.32*	8.28	*1.35*	13.92	*1.14*
binary	0.28	*1.00*	2.35	*0.98*	3.51	*1.02*	1.38	*1.15*	4.35	*1.03*	7.58	*1.24*	12.67	*1.04*
stream	0.28	*1.00*	-	-	-	-	2.84	*2.36*	16.92	*4.01*	1.17	*0.19*	11.54	*0.95*
debug	10.34	*37.27*	33.40	*13.89*	45.14	*13.09*	18.68	*15.54*	34.21	*8.10*	34.65	*5.67*	54.09	*4.44*

Figure 3.6: Latency of Coupling Techniques - Table

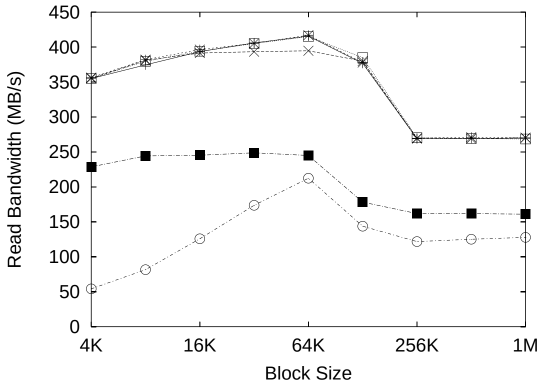
This table shows the same data as Figure 3.5. For each coupling technique and system call, the call latency is given (μs) along with the slowdown (α) relative to the unmodified case. For clarity, the variance in measurement shown in the previous figure is omitted. Note that the stream interface does not include the `stat` *or* `open/close` *calls.*

The unmodified case gives the performance of this benchmark without any agent attached, while the remaining five show the same benchmark modified by each coupling technique. In each case, a minimal agent is used to trap each system call and then re-execute the call without modification. Note that the streaming case is not included in the `stat` and `open-close` calls, because these operations are not found in the stream interface.

As can be seen, the static linking and binary rewriting techniques add no measurable latency to system calls. The dynamic method has overhead on the order of several microseconds, as it must manage the active layer infrastructure and perform at least one indirect jump. This overhead causes a slowdown of 1-6x, depending on the call. Streaming has an interesting property: a small write to a stream is *faster* than writing to an ordinary file. This highly optimized code path in the operating system allows for the rapid deposit of several bytes into a buffer without the expense of navigating file system structures such as inodes and indirect blocks. However, reads larger than the stream buffer size (typically fixed at 4KB) are slower than reads to the filesystem, because such reads require multiple context switches between the agent and the program. The debugger has the greatest overhead of all the techniques by a sizable margin, slowing down each system call between 4.4x and 37x times. This is due to the large number of context switches depicted in Figure 3.3.a.



(a) Write Bandwidth



(b) Read Bandwidth

Figure 3.7: Bandwidth of Coupling Techniques

These graphs show the maximum bandwidth achieved by each technique writing to and reading from the filesystem. Note that both streaming and debugging pay a price in bandwidth because each makes an extra data copy. The debugger is especially sensitive to block size because of the latency shown in Figure 3.5.

Figure 3.7 shows the data transfer bandwidth achievable with each technique. This figure was generated with the same experimental setup as before, but measuring the available bandwidth when transferring 100MB of data to or from disk with varying transfer block sizes. To avoid measuring the vagaries of periodic buffer cache flushes and other uncontrollable events, each point represents the *maximum* of ten measurements. As before, each agent simply traps the operations in question and executes them without modification.

The three internal techniques impose no bandwidth overhead at all compared to the unmodified case. Both of the external techniques do incur a fair bandwidth overhead, depending on the block size used in the transfer. Smaller block sizes exaggerate the effect of the system call overhead on data transfer, while larger block sizes diminish opportunities for overlap between agent and program operation. The balance between these concerns occurs in both the read and write cases at a block size of 64KB. In the write case, streaming achieves about 80 percent of the unmodified bandwidth, while debugging achieves about 75 percent. In the read case, the figures are about 60 and 50 percent, respectively.

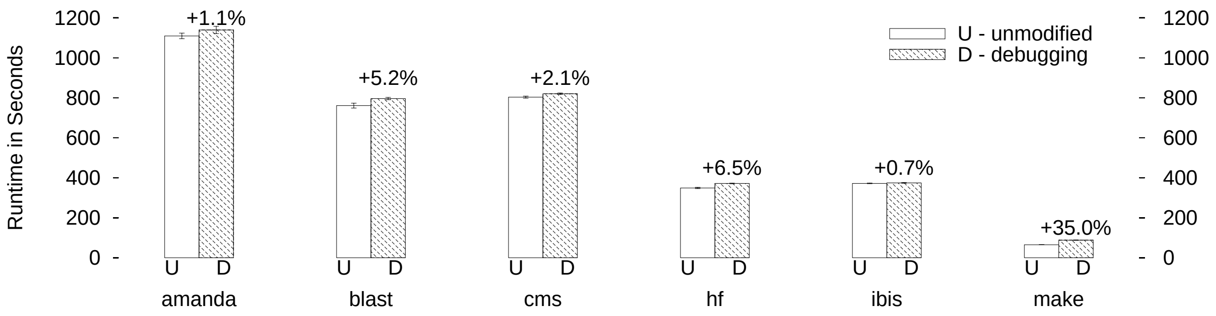


Figure 3.8: Overhead of Debugging on Applications

This figure shows the total runtime overhead of the debugging technique on the six applications introduced in Chapter 1. The other techniques were unable to couple to these applications. Note that the overhead is significantly less than data latency and bandwidth overheads shown in the previous figures.

Because the transfer block size has such a dramatic effect on achievable bandwidth for the debugging technique, it is important to find ways to coerce applications into using the appropriate block size. Fortunately, such a mechanism is already present. Conventional applications using the standard library make use of a block size "hint" expressed through the `stat` system call. A debugger agent simply modifies this hint to be 64 KB, and thus most applications make use of the optimal block size.

Despite the significant overhead of debugging on these microbenchmarks, the cumulative effect on real applications is relatively modest. Figure 3.8 shows the overhead of running the five candidate applications described in Chapter 2 on the same system as above, both unmodified and coupled to a debugging agent. Each bar shows the mean and variance of ten measurements. The percent overhead due to the debugging agent is shown above the bars for each application.

The overhead in runtime for the candidate applications ranges from less than 1 percent (`ibis`) up to 6.5 percent (`hf`.) However, the comparison application (`make`) is much harder hit by this technique and is slowed down by 35 percent.

The reason for the disparity between `make` and the scientific applications is shown in

	open	read	write	mmap	seek	stat	brk	time	other	total	rate
ama.	5034	9873	117771	518	32	1702	122831	10039	4411	272211	**103**
blast	3524	1108	186267	2023	3	4719	28178	2501	4547	232870	**131**
cms	268	76016	3892	98	76009	315	91	251	615	157555	**110**
hf	505	9760	152271	71	149130	665	1092	25	1505	315024	**343**
ibis	1192	590	926	169	1943	1427	332	3	1462	8044	**10**
make	88255	17041	4329	18315	12905	47878	92139	440	45891	327193	**1012**

Figure 3.9: System Calls by Application

This figure details the actual system calls performed by each application. The final column shows the number of system calls performed per second. Note that `make` *performs an order ot magnitude more system calls per second than the other programs.*

Figure 3.9, which summarizes the total number of system calls made by each application. `make` stands out in its use of non-data-carrying system calls such as `open` and `stat`. The `make` program itself must extensively scan directories, but also the invocation of each subprocess and script invoked by `make` makes use of `open` and `stat` in order to search for libraries during the dynamic linking phase of each newly created process. The **rate** column shows the number of system calls made per second of runtime, and `make` exceeds the other applications by an order of magnitude. This reliance on metadata (also observed by Spasojevic [135]) is an important distinction between scientific and interactive workloads.

These overheads are likely to be acceptable in the context of high-throughput distributed batch computing. If the deployment of an agent enables the user to harness twice as many machines as without, then a per-job slowdown of several percent is irrelevant. Of course, if the user gains no benefit from using an agent, it is best to leave out any unneeded overhead.

The reader has surely noticed that I have omitted a number of techniques from this measurement. The reason is that I was unable to employ them on these applications! The very nature of the applications prevented this. All involved multiple processes and scripting languages, some were statically linked, and some did not provide source code. For all of the complications described above, employing these techniques did not work. Of course, it may be *possible* to do so, given sufficient manpower and source access, but a user's performance requirements must be quite stringent to make this effort worth a small reduction in overhead.

3.6 Conclusion

I have described a variety of job coupling techniques and evaluated low-level performance, general usability, and high-level performance on scientific applications. Each technique varies in usability and cost.

The polymorphic extension is the appropriate technique to use when an application consists of a small number of programs whose sources are available for modification. No special actions need be taken in order to understand the interface between the job and the agent. The compiler may be relied upon to enforce typechecking and other language constraints. Little or no performance penalty is paid.

Although internal agents are fast and attractive, they create an extraordinary maintenance problem. The essential difficulty is that the binary interface between an application and the standard library is poorly specified and in constant flux. Although all manner of standards bodies have legislated the source-level interface between programs and the standard library, source constructs may have little relationship with the binary reality: the names and structures of interest to the application programmer are often transformed into something complex and undocumented at the library interface.

The debugger is the appropriate technique when many distinct programs are to be used, or when the source for an application is not available. Although the debugger has its complexities, the interface between a process and the operating system is a well-defined and relatively static across operating system upgrades. Holes in the agent are readily identified and fixed. Although there is a performance penalty to be paid, it is not enormous in the realm of scientific applications.

What technique should be chosen? In general, I recommend the use of the debugger trap because it is instantly effective on all applications without any extra work. For some applications, particularly those that are metadata intensive, the increased latency of system calls may be unacceptable. In this case, an internal agent may be appropriate if the cost of maintaining such an agent can be offset by the speedup it obtains.

Chapter 4

Data Coupling

4.1 Introduction

Using the experience gained with the various job coupling techniques in the last chapter, I have built a general-purpose agent called **Parrot** [147] for attaching standard Unix applications to distributed I/O systems. Such systems are presented to programs as ordinary entries in the filesystem. Parrot has many uses in a distributed computing environment:

Seamless integration. The construction of a new type of storage protocol or device is frequently accompanied by the construction of tools to visualize, organize, and manipulate the contents. Building such tools is both time consuming and wasteful, as many tools already exist for these tasks on standard filesystems. As shown in Figure 4.1, Parrot enables ordinary tools to browse a remote archive, in this case the hierarchical mass storage server (MSS) at the National Center for Supercomputing Applications (NCSA).

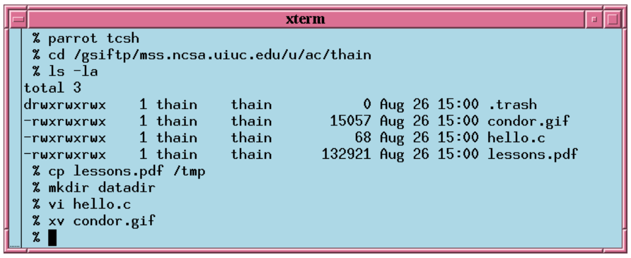


Figure 4.1: Interactive Browsing

Improved reliability. Naturally, the networked services that are accessed by an agent are far less reliable than a local filesystem. Remote services are prone to failed networks, power outages, expired credentials, and many other problems not found in local system services. An agent can attach an application to a service with improved reliability. Parrot can be used to add reliability at the filesystem layer by detecting and repairing failed I/O connections. A similar idea is found in Rocks [155], which provides a reliable TCP connection despite network outages and address changes.

Private namespaces. Batch applications are frequently hardwired to use certain file names for configuration files, data libraries, and even ordinary inputs and outputs. By specifying a private namespace for each application instance, many may be run simultaneously while keeping their I/O activities separate. For example, ten instances of an application hardwired to write to `output.txt` may be redirected to write to `output.n.txt`, where n is the instance number. A private namespace may also be constructed for performance concerns. A centralized data server can only serve so many simultaneous remote applications before it becomes saturated. If copies of necessary data are available elsewhere in a system, a private namespace may be constructed to force an application to use a nearby copy. Whether for logical or performance purposes, a private namespace can be built at many points in the lifetime of an application. It may be fixed throughout the program's lifetime or it may be resolved on demand as the program runs by an external service.

Remote dynamic linking. Dynamic linking presents several problems in naming and execution. A majority of standard applications are linked against dynamic libraries that are named and loaded at runtime. Dynamic linking reduces the use of storage and memory by allowing applications to share routines. However, this advantage can become a liability of complexity, perfectly captured by the popular phrase "DLL hell." External coupling techniques permit a remotely-executing application to fetch all of its libraries from a trusted source as they are needed. Such libraries may then be shared normally at the execution site without burdening the end user.

Profiling and debugging. The vast majority of applications are designed and tested on standalone machines. Surprises occur when such applications are moved into a distributed system. Both the absolute and relative cost of I/O operations change, and techniques that were once acceptably inefficient (such as unbuffered writes) may become disastrously inefficient. By attaching an agent to the application, the user may easily generate a trace or summary of the I/O behavior and observe precisely what the application does. Just such a tracing technique was used to produce a detailed study [143] of the five candidate applications introduced in Chapter 1.

4.2 Architecture

Internally, Parrot is a library for performing Unix-like I/O on remote data services. It provides an interface with entry points like `parrot_open` and `parrot_read`. An application may be written or modified to invoke the library directly, or it may be attached via the various coupling techniques described in the previous chapter. For the reasons already described, I will concentrate on the use of Parrot by way of the debugging interface.

The internal structures of Parrot, shown in Figure 4.2, bear a strong resemblance to the file structures in a conventional operating system. Parrot tracks a family tree of processes, recording a table of open file descriptors, seek pointers, and similar device-independent structures for each. At the lowest layer are a series of device drivers that implement access to remote I/O system. Unlike an operating system, Parrot does not know the structure of remote devices at the level of inodes and blocks. It refers to remote open files by name, and may multiplex many applications' I/O requests through one remote channel.

Parrot has a large number of entry points for I/O operations. We may classify them into two categories: operations on file descriptors and operations on file names. The former

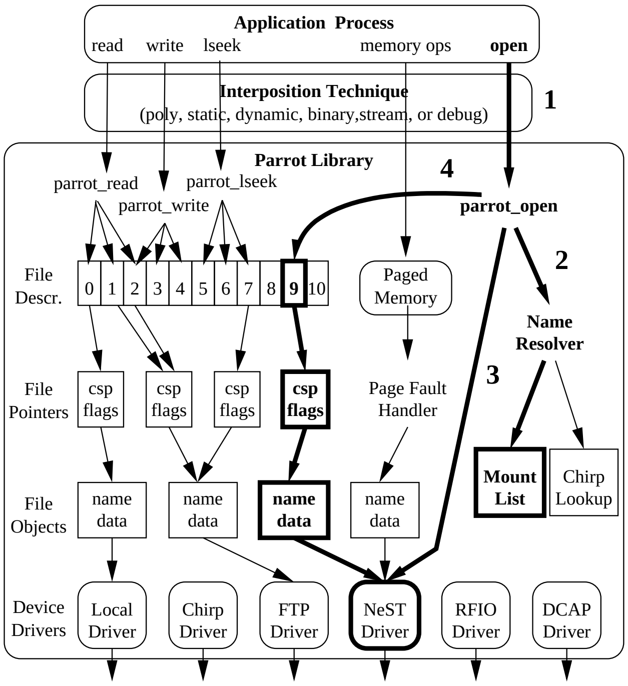


Figure 4.2: Architecture of Parrot

This figure shows the architecture of Parrot. Many of the data structures correspond closely to the file structures in a standard operating system. The bold lines show the steps needed to open a file. 1: Trap open system call. 2: Consult name resolver. 3: Open file via driver. 4: Install data structures.

traverse most of the data structures in Parrot, while the latter take a more direct route to the device drivers.

Operations such as `read`, `write`, and `lseek` operate on file descriptors. Upon entering Parrot, these commands check the validity of the arguments, and then descend the various data structures. Operations `read` and `write` examine the current file pointer and use it as an argument to call a `read` method in the corresponding file object. The file object, through the device driver, performs the necessary remote operation. Other operations such as `rename`, `stat`, and `delete` operate on file names. Upon entering Parrot, these commands first pass through the *name resolver*, which may transform the program-supplied name(s) according to a variety of rules and systems. The transformed names are passed directly to the device driver, which performs the operation on the remote system.

The bold outlines in Figure 4.2 show how an `open` system call consults the name resolver, opens a file using a protocol driver, and then installs all of the data structures to represent the open file.

Most Unix applications access files through explicit operations such as `read` and `write`. However, files may also be memory-mapped, particularly dynamically linked libraries. In a standard operating system, a memory mapped file is a separate virtual memory segment whose backing store is kept in the file system rather than in the virtual memory pool. Parrot accomplishes the same thing using its own underlying drivers, thus reducing memory mapped files to the same mechanisms as other open files.

Memory-mapped files are supported in one of two ways, depending on the coupling technique in use. If Parrot is attached using an internal technique, then memory mapped files may be supported by simply allocating memory with `malloc` and loading the necessary data into memory by invoking the necessary device driver. As a matter of policy, the entire file can be loaded when `mmap` is invoked, or it can be paged in on demand by setting the necessary memory protections and trapping the software interrupts generated by access to that memory. If Parrot is attaching using the debugger technique, then the entire file is loaded into the I/O channel, and the application is redirected to `mmap` that portion of the channel. Parrot does not currently have any write mechanism or policy for memory-mapped files, as I have yet to encounter any application that would require it.

Parrot has two buffering disciplines: fine-grained and whole file. By default, Parrot simply performs fine-grained partial file operations on remote services to access the minimal amount of data needed to satisfy an application's immediate `read`s and `write`s. I have taken this route for several reasons. First, whole-file fetching introduces a large latency when a file is first opened. This latency is often an unnecessary price when an application could take advantage of overlapped CPU and I/O access by reading streamed files sequentially.

Second, few remote I/O protocols have a reliable mechanism for ensuring synchronization between shared and cached files; we do not wish to introduce a new synchronization problem. Finally, a variety of systems have already been proposed for managing wide area replicated data [28, 125, 23]. I prefer to make Parrot leverage such systems via fine-grained access protocols rather than implement replica management anew.

Optionally, Parrot may perform whole-file staging and caching upon first `open` in a manner similar to that of AFS [68] and UFO [10]. Once this long latency is paid, a file may be accessed efficiently in local storage. Protocols that only provide sequential access, such as FTP, require the use of the cache to implement random access. At each `open`, a cached file is validated by performing a remote `stat` to determine if the cached copy is up to date. If the file's size or modification time has changed, then it is re-fetched. Further, whole-file fetching is necessary in order to implement the `exec` call on a remotely stored program, regardless of the protocol.

4.3 Protocols and Semantics

Parrot is equipped with a variety of drivers for communicating with external storage systems; each has particular features and limitations. The simplest is a local driver, which simply passes operations on to the underlying operating system. The Chirp protocol was designed specifically for use with Parrot: it corresponds very closely to the Unix interface. A standalone Chirp server is distributed with Parrot and allows an unprivileged user to establish a secure file server. The venerable File Transfer Protocol (FTP) [112] has been in heavy use since the early days of the Internet. Its simplicity allows for a wide variety of implementations, which, for our purposes, results in an unfortunate degree of imprecision which I will expand upon below. Parrot supports the secure GSI [11] variant of FTP. The NeST protocol is the native language of the NeST storage appliance [26], which provides an array of au-

	name binding	discipline	dirs	metadata	symlinks	connections
unix	open/close	random	yes	direct	yes	-
chirp	open/close	random	yes	direct	yes	per client
ftp	get/put	sequential	varies	indirect	no	per file
nest	get/put	random	yes	indirect	yes	per client
rfio	open/close	random	yes	direct	no	per file/op
dcap	open/close	random	no	direct	no	per client

Figure 4.3: Protocol Compatibility with Unix

thentication, allocation, and accounting mechanisms for storage that may be shared among multiple transient users. The RFIO and DCAP protocols were designed in the high-energy physics community to provide access to hierarchical mass storage devices such as Castor [22] and DCache [48]. This selection of protocols does not include every known storage device, but does provide an interesting cross-section of semantics for discussion purposes.

Because Parrot must preserve Unix semantics for the sake of the application, the foremost concern is the ability of each of these protocols to provide the necessary semantics. Performance is a secondary concern and we will see below that it is affected by semantic issues. A summary of the semantics of each of these protocols is given in Figure 4.3.

In Unix, name binding is based on a separation between the namespace of a filesystem and the file objects (i.e. inodes) that it contains. This is known as an open/close model. The `open` system call performs an atomic binding of a file name to a file object, which allows a program to lock a file object independently of the renaming, linking, or unlinking of names that point to it. This model is reflected in the Chirp, RFIO, and DCAP protocols, which all provide distinct open/close actions separately from data access. Unlike Unix, FTP and NeST have a get/put model which performs a name lookup at every data access. In the get/put model, an application may lose files or become confused if the file namespace is manipulated by another process while files are in use.

With the exception of FTP, all of the protocols provide inexpensive random (non-sequential) access to a file without closing and re-opening it. Random access permits the efficient manipulation of a small portion of a large remote file without retrieving the whole thing. The sequential nature of FTP requires that Parrot make local copies of such files in order to make changes and then replace the whole file. Some variants of FTP allow for bulk reads and writes to start at an arbitrary file offset, however these operations are not universally supported.

Directories are supported completely by Chirp, NeST, and RFIO; one may create, delete and list their contents. DCAP does not currently support directory access, because it is typically used alongside a kernel-level NFS client for metadata access. Support for directories in FTP varies greatly. Although the FTP standard mandates two distinct commands for directory lists, LIST and NLST, there is little agreement on their proper behavior. LIST provides a completely free-form text dump that is readable to humans, but has no standard machine-readable structure. NLST is meant to provide a simple machine-readable list of directory entries, but we have encountered servers that omit subdirectory names, some that omit names beginning with dot (.), some that insert messages into the directory list, and even some that do not distinguish between empty and non-existent directories.

Most metadata is communicated in the Unix interface through the `stat` structure returned by the `stat`, `fstat`, and `lstat` system calls. Chirp, RFIO, and DCAP all provide direct single RPCs that fill this structure with the necessary details. FTP and NeST do not have single calls that provide all this information, however, the necessary details may be obtained indirectly through multiple RPCs that determine the type, size, and other details one by one.

Only Chirp and NeST provide support for managing symbolic links. This feature might be done without, except that remote I/O protocols are often used to expose existing filesystems that already contain symbolic links. A lack of symbolic links at the protocol level can result in confusing interactions for programs as well as the end user. For example, a symbolic link may appear as in an FTP directory listing, but, without explicit operations for examining links, it will appear to be an inaccessible file with unusual access permissions.

Finally, the connection structure of a remote I/O protocol has implications for semantics as well as performance. Chirp, NeST, and DCAP require one TCP connection between each client and server. FTP and RFIO require a new connection made for each file opened. In addition, RFIO requires a new connection for each operation performed on a non-open file. Because most file system operations are metadata queries, ordinary activity can result in an extraordinary number of connections in a short amount of time. Even ignoring the latency penalties of this activity, a large number of TCP connections can consume resources at clients, servers, and network devices such as address translators.

4.4 Chirp Protocol

The Chirp protocol will be used extensively in the following pages, so it will be useful to explore it in some detail here. Chirp was designed specifically for use with Parrot and Condor, and corresponds fairly closely to the Unix I/O interface. There are currently two implementations of the protocol. The first consists of a C client and server that are distributed with Parrot. The second consists of a Java client and C++ server integrated into Condor. The former is the more complete implementation and has all the features described here. The latter will be discussed in Chapter 5.

To initiate a conversation, a Chirp client connects to a Chirp server via TCP. One connection is maintained for all data between a client and server, regardless of the files opened or accessed. Once connected, the client must authenticate itself to the server. The current implementation has a negotiation feature that allows the client to authenticate itself by several methods, including Kerberos [137], GSI [53], shared secrets, or network addresses. The authentication protocol is not strictly part of Chirp, and will not be discussed further.

Chirp is a request-response protocol. The client initiates all activity by sending a com-

Operations on Open Files

Name	Arguments	Results
open	(path,flags,mode)	→ (fd,stat)
close	(fd)	
fchmod	(fd,mode)	
fchown	(fd,uid,gid)	
fstat	(fd)	→ (stat)
fstatfs	(fd)	→ (statfs)
fsync	(fd)	
ftruncate	(fd,length)	
pread	(fd,length,offset)	→ (data)
pwrite	(fd,data,length,offset)	

Special Operations

Name	Arguments	Results
canonicalize	(path)	→ (newpath)
getdir	(path)	→ (dirlist)
getfile	(path)	→ (stat,data)
lookup	(path)	→ (newpath,lifetime)
putfile	(path,data,mode,length)	

Operations on File Names

Name	Arguments	Results
access	(path,mode)	
chmod	(path,mode)	
chown	(path,uid,gid)	
lchown	(path,uid,gid)	
link	(path,newpath)	
lstat	(path)	→ (stat)
mkdir	(path,mode)	
readlink	(path)	→ (linkpath)
rename	(path,newpath)	
rmdir	(path)	
stat	(path)	→ (stat)
statfs	(path)	→ (statfs)
symlink	(path,newpath)	
truncate	(path,length)	
unlink	(path)	
utime	(path,atime,mtime)	

Figure 4.4: Chirp Protocol Summary

This figure summarizes all of the operations in the Chirp protocol using an abstract syntax. The name, arguments, and results (if any) of each operation are shown. Each operation also returns a code indicating success or failure, which is not shown. Most operations on open files and filenames correspond very closely to operations in Unix. Several special operations that do not correspond to Unix operations are described in the text.

mand code, the arguments to the command, and then waiting for a response from the server. A good implementation of the protocol (as found in Parrot) allows for the client to give up if a response is delayed for a configurable amount of time. Most commands in the protocol involve a small amount of data and can be implemented as ordinary remote procedure calls.

Figure 4.4 summarizes all of the commands in the Chirp protocol. Each command has a unique name, a list of arguments, and a list of results. Each command also has an integer result (not shown) that indicates success or the reason for a failure. Most commands correspond closely to similarly named counterparts in Unix. For example, the `open` command requests access to a file by name and yields an integer file descriptor. (Unlike Unix, `open` also returns the file's metadata in the form of a `stat` structure.) The file descriptor may then be used to read, write and perform other Unix-like commands on the open file. When the file is no longer needed, the `close` command releases the file descriptor. Just as in Unix, a variety of commands such as `access` and `chmod` operate on files by name.

All filenames in the Chirp protocol are relative to the root of the filesystem at the Chirp server. The server does not track the current working directory of the client, because the client may be representing multiple processes or may have its current working directory on another server altogether.

Several commands do not have exact Unix counterparts. The `canonicalize` command queries for the existence of a file or directory at the server, and returns its full pathname with all intermediate symbolic links evaluated and removed. This is an optimization that allows a client to easily obtain the full name of a directory in order to satisfy system calls such as `getcwd`. The `getdir` command atomically obtains a complete directory listing, while `getfile` and `putfile` stream whole files to and from the server. The `lookup` command performs a filename mapping in a remote mountlist, as described in Section 4.5.

The Chirp protocol may be thought of as *semi-stateful*. A Chirp *server* is stateful: It tracks the set of files that a client has opened. If the connection between the client and server should be lost, the server is responsible for closing those files. However, a client can render the connection effectively stateless: If a client also tracks the set of files that it has opened, it may recover open files after a discussion. This possibility is discussed in Section 4.6.3.

The close-on-failure property makes the Chirp protocol naturally suited to a multi-process implementation. In the current C implementation, each incoming Chirp connection causes a new handling process to be forked. The handling process changes privilege level (if necessary) and then serves incoming Chirp commands by executing their Unix equivalents. If the connection is lost, the handling process may simply exit, and the client's files will be closed automatically by the local operating system.

4.5 Name Resolution

Parrot's name resolver allows the user to construct a custom namespace appropriate for each application. In the simplest case, the name resolver takes no action, and an application may explicitly operate on the global file space. Applications may specify ordinary local file names such as `/etc/passwd` or fully-qualified remote filenames such as `/ftp/ftp.cs.wisc.edu/RoadMap`.

Alternatively, a user may provide a *mountlist*, a file that maps logical file names and directories to specific physical devices, much like the Unix file `/etc/fstab`. Here is a sample mountlist that directs an application to use input data from a well-known FTP archive, write outputs to a local Nest server, and make use of dynamic libraries exported by a Chirp server on a reference machine:

```
/data    /ftp/ftp.experiment.org/expt1/archive
/output  /nest/archive.cs.wisc.edu/outputs
/lib     /chirp/rh7.cs.wisc.edu/lib
```

The name resolver is a natural place for attaching an application to shared naming systems. A naming system is useful when a given document may be replicated across a system and an application simply needs to access the closest available one. A variety of systems such as the the Replica Location Service [149] and the Handle System [140] are examples of shared location services. A Chirp server can also serve as remote name resolver, as shown in Figure 4.5. Each reference by the job to a filename results in a `lookup` call to the Chirp server, which consults a mountlist and returns a result. This shared service allows definition to control the resources consumed by a large set of running jobs. As available storage resources change, a single mountlist can be updated to reconfigure the entire system. Of course, this agility comes at a cost: every reference to a file name results in

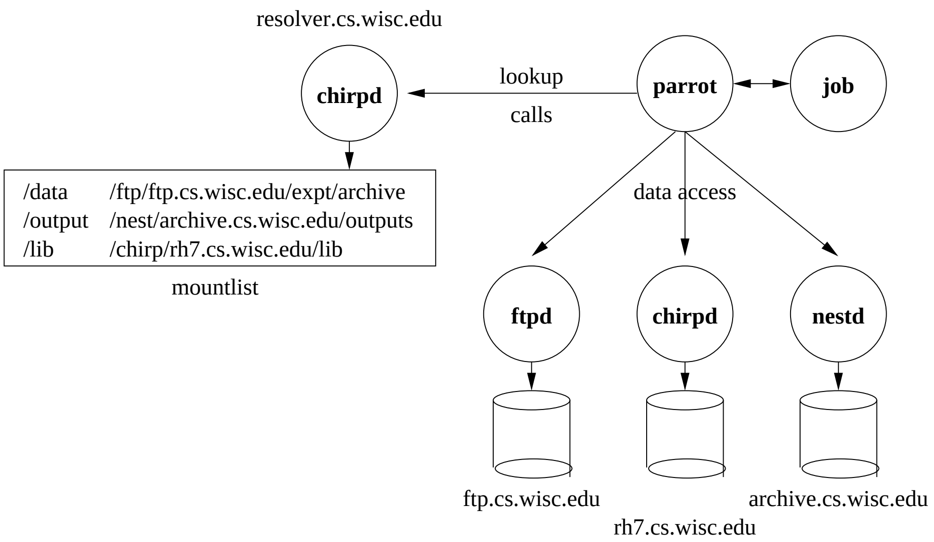


Figure 4.5: Name Resolution via a Chirp Server
This figure shows a running job making use of the remote name resolution feature in Chirp. Whenever a job accesses a file by name, Parrot issues a `lookup` *Chirp call to the server* `resolver.cs.wisc.edu` *in order to consult the remote mountlist.*

a network operation. To allow the user to strike a balance between agility and performance, the response from the Chirp server includes the lifetime of the result, allowing Parrot to cache lookups on behalf of a job.

Because Parrot manages access to the entire file namespace through one control point, it can also serve as a sandbox in order to prevent a buggy or malicious job from accessing or modifying portions of the namespace. A complete language for specifying the parameters of a sandbox for interactive applications may be found in tools such as Janus [61].

For our purposes, it is sufficient to allow cutting off portions of the namespace with a simple syntax. For example, the following addition to a mountlist would prevent any access to the local temporary filesystem on a remote machine:

```
/tmp    DENY
```

4.6 Error Handling

Error propagation has not been a serious obstacle in the design of traditional operating systems. As new models of file interaction have developed, attending error modes have been added to existing systems by expanding the software interface at every level. For example, the addition of distributed file systems to the Unix kernel created the new possibility of a stale file handle, represented by the `ESTALE` error. As this error mode was discovered at the very lowest layers of the kernel, the value was added to the device driver interface, the file system interface, the standard library, and expected to be handled directly by applications.

There is no such luxury in an interposition agent. Applications use the existing interface, and we have neither the desire nor the ability to change it. Yet, the underlying device drivers generate errors ranging from the vague “file system error” to the microscopically precise “server’s certification authority is not trusted.” How should the unlimited space of errors in the lower layers be transformed into the fixed space of errors available to the application?

In answering this question, we must keep in mind that the agent is not the last line of defense. Together, the agent and the job live in a larger context, supervised by the surrounding batch system. In this context, the agent may appeal to the batch system to take some higher-level scheduling action. This is not to say that we should always pass the buck to the batch system. Rather, we must perform triage:

Transformable errors may easily be converted into a form that is both honest and recognizable by the application. Such errors are converted into an appropriate `errno` and passed up to the application in the normal way. Transforming an error may require additional effort or communication on behalf of the agent.

Permanent errors indicate that the process has a fatal flaw and cannot possibly run to completion. With this type of error, Parrot must halt the process in a way that makes it clear the batch system must not reschedule it.

Escaping errors are neither transformable nor permanent. An escaping error indicates that the process cannot run here and now, but has no inherent flaw. An agent's proper response to an escaping error is to indicate to the CPU management system that the job is to release the CPU, but would like to execute later and retry the I/O operation. Chapter 5 will address escaping errors in detail.

Each of the three types of errors – transformable, permanent, and escaping – come from two distinct sources of errors – a mismatch of requests, or a mismatch of results. A mismatch of requests occurs when the target system does not have the needed capability. A mismatch of results occurs when the target system is capable, but the result has no obvious meaning to the application. Let us consider each in turn.

4.6.1 Mismatched Requests

The first difficulty comes when a device driver provides no support whatsoever for an operation requested by the application. There are three different solutions to this problem, depending on the expectation of the application's ability to handle an error. Representative examples are `getdents`, which retrieves a directory listing, `lseek`, which changes a file's seek pointer, and `stat`, which obtains a file's metadata.

Some I/O services, such as DCAP, do not permit directory listings. A call to `getdents` cannot possibly succeed. Such a failure may be trivially represented to the calling application as "permission denied" or "not implemented" without undue confusion. Applications understand that `getdents` may fail for any number of other reasons on a normal filesystem, and are thus prepared to understand and deal with such errors.

In contrast, almost no applications are prepared for `lseek` to fail. It is generally understood that any non-terminal file may be accessed randomly, so few (if any) applications even bother to consider the return value of `lseek`. If we use `lseek` on an FTP server without local caching enabled, we risk any number of dangers by allowing a never-checked command to fail. Therefore, an attempt to seek on a non-seekable file results in a permanent error with a message on the standard error stream.

The `stat` command offers the most puzzling difficulty of all. `stat` simply provides a set of meta-data about a file, such as the owner, access permissions, size, and last modification time. The problem is that few remote storage systems provide all, or even most, of this data. For example, FTP provides a file's size, but no other meta-data in a standard way.

An agent might cause `stat` to report "permission denied" on such systems, under the assumption that brutal honesty is best. Unfortunately, this causes all but the most trivial of programs to fail. All manner of code, including command-line tools, large applications, and the standard C library, invoke `stat` on a regular basis. At first glance, it appears that the necessary information simply cannot be extracted from most remote I/O systems. However, we may construct a workaround by surveying the actual uses of `stat`:

- **Cataloging.** Commands such as `ls` and program elements such as file dialogs use `stat` to annotate lists of files with all possible detail for the interactive user's edification.

- **Optimization.** The standard C library, along with many other tools, uses `stat` to retrieve the optimal block size to be used with an I/O device.

- **Short-circuiting.** Many programs and libraries, including the command-line shell and the Fortran standard library, use `stat` or `access` to quickly check for the presence of a file before performing an expensive `open` or `exec`.

- **Unique identity**. Command line tools use the unique device and file numbers returned by `stat` to determine if two file names refer to the same physical file. Unique identifiers used to prevent accidental overwriting and recursive operations.

In each of these cases, there is very little harm in concocting information. No program can rely on the values returned by `stat` because it cannot be done atomically with any other operation. If a program uses `stat` to measure the existence or size of a file, it still must be prepared for `open` or `read` to return conflicting information. Therefore, we may fill the response to `stat` with benevolent lies that encourage the program to continue for both reading and writing. Each device driver fills in whatever values in the structure it is able to determine, perhaps using multiple remote operations, and then fills the rest with defaults. For example, Parrot simulates inode numbers by computing a hash of the file's full path. The last modification time may be set to the current time. The file system block size is simply set to a tunable value, as described in Chapter 3.

4.6.2 Mismatched Results

Several device drivers have the necessary machinery to carry out all of a user's possible requests, but provide vague errors when a supported operation fails. For example, the FTP driver allows an application to read a file via the GET command. However, if the GET command fails, the only available information is the error code 550, which encompasses almost any sort of file system error including "no such file," "access denied," and "is a directory." The Unix interface does not permit a catch-all error value; it requires a specific reason. Which error code should be returned to the application?

One technique for dealing with this problem is to interview the service in order to narrow down the cause of the error in a manner similar to that of an expert system. Figure 4.6 shows the interview tree for a GET operation. If the GET should fail, Parrot assumes the named file is actually a directory and attempts to move to it. If that succeeds, the error is "not a file." Otherwise, it attempts to SIZE the named file. If that succeeds, the file is present but inaccessible, so the error is "access denied." If it fails, the error is finally "no

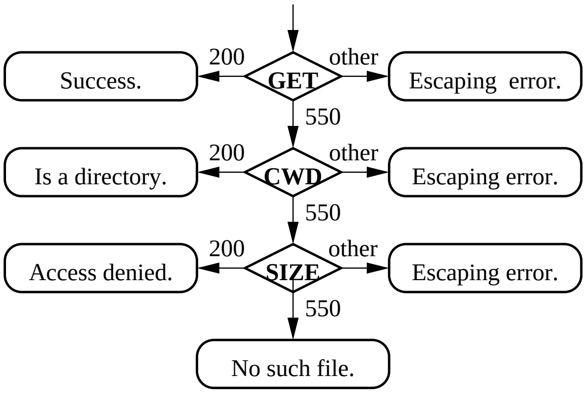


Figure 4.6: An Error Interview

An error interview for retrieving a file via FTP. If the initial GET operation should fail, the interview continues with CWD and SIZE operations. At each attempt, a successful answer yields a specific error, while the non-specific error 550 continues the interview. Any other response results in an escaping error.

such file."

The error interview technique also has some drawbacks. It significantly increases the latency of failed operations, which are quite common in shells and scripting languages that may search for a file in many places. In addition, the technique is not atomic, so it may determine an incorrect value if the remote filesystem is simultaneously modified by another process.

4.6.3 Disconnection

Any I/O system must be prepared for the possibility of a network interruption between a client and server. Parrot can deal with such disconnections, but as with everything else, the details of the protocol have semantic consequences for the application.

When Parrot discovers that it has been disconnected from a server, it attempts to retry the connection at exponentially increasing intervals until it reaches a configurable time limit. (An infinite time limit is equivalent to the NFS concept of "hard-mounting.") If the time limit is reached without reconnection, then Parrot emits an escaping error. However, the more interesting case is what happens if the reconnection is successful.

Reconnection to a `get/put` service is easy. Because such servers maintain no state about clients, Parrot can simply rebuild the TCP connection, and resume exactly where it left off. Of course, it is possible that another program has moved or deleted the files Parrot was using, but this was already possible regardless of the disconnection. Such services do not provide Unix semantics in any case.

Reconnection to an `open/close` service is more complicated, precisely because servers track what files a client has open in order to provide Unix semantics. In this case, Parrot makes the best effort it can to reconstruct the client's state. Upon reconnection, Parrot simply re-opens all of the files that were open before the disconnection and examines their inode numbers. (This is why the Chirp `open` call returns a `stat` structure in addition to the file descriptor.) If the files cannot be opened or the inodes have changed, then the client's state is irretrievable and Parrot must emit an escaping error.

It is worth noting a failure mode that was unexpected in design but is common in practice. At the time of writing, many hosts are connected to the Internet by way of a Network Address Translation (NAT) device that multiplexes a single public IP address between multiple machines, giving each a private address and a transparent TCP proxy. Because the NAT device has limited memory and limited knowledge of the hosts that communicate through it, it must periodically discard TCP connections that it believes to be idle.

This seemingly innocuous design was initially fatal to many batch jobs that used Parrot to access data on a remote host on the other side of the NAT. Typically, batch jobs would access data for a short time and then compute in silence for minutes or hours. The NAT would discard the idle connection while the job was computing. When the job later decided to perform I/O via Parrot, it would attempt to use the discarded TCP connection, but the NAT would silently reject all IP packets on that connection, not even responding with a

TCP reset packet. The result was that both the client and server would sit idle for hours waiting for the last-ditch TCP timeouts.

The solution to this problem is to ignore the TCP connection itself as a signal of connection liveness. Instead, the I/O protocol itself must serve as a keepalive message. If no response to an I/O request is received within a moderate amount of time, then the TCP connection is discarded with prejudice and the recovery protocol is initiated right away. In short, Parrot must be aggressive in detecting failure in order to operate across the NAT devices that are now common on the Internet.

4.6.4 Everything Else

There remains a very large space of infrequent errors that simply have no expression at all in the application's interface. A NeST might declare that a disk allocation has expired and been deleted. An FTP server may respond that a backing store is temporarily offline. User credentials, such as Kerberos or GSI certificates, may expire, and no longer be valid. In response, we may reallocate space, rebuild connections, or attempt to renew certificates. Like all the other error cases, these tactics take time, resources, and have no guarantee of eventual success. After some (configurable) effort, Parrot simply gives up and emits an escaping error.

4.7 Performance

I have deferred a discussion of performance until this point in order to demonstrate the performance effects of semantic constraints. Although it is possible to write applications explicitly to use remote I/O protocols in the most efficient manner, Parrot must provide conservative and complete implementations of Unix operations. For example, an application may only need to know if a file exists. If it requests this information via `stat`, Parrot is

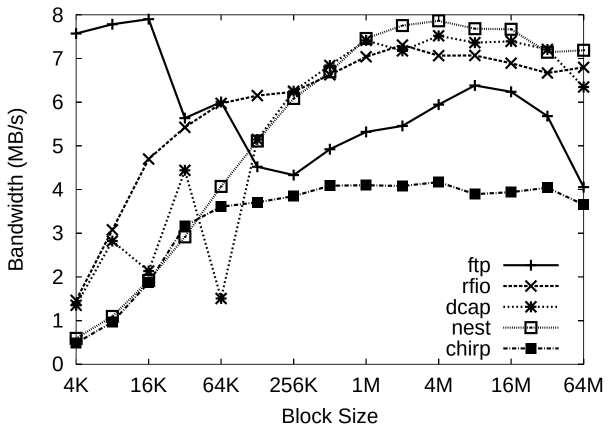


Figure 4.7: Write Throughput by Protocol
This figure shows the maximum throughput achieved over a local area network by each I/O protocol via Parrot. Note that all protocols are sensitive to block size. The hiccup in DCAP at 64K is described in the text.

obliged to fill the structure with everything it can, possibly at great cost.

The I/O services discussed here, with the exception of Chirp, are designed primarily for efficient high-volume data movement. Figure 4.7 compares the throughput of the protocols at various block sizes. The throughput was measured by copying a local 128 MB file into the remote storage device with the standard `cp` command equipped with Parrot and a varying default block size, as controlled through the `stat` emulation described above. As with previous throughput measurements, each was repeated 10 times, and the *maximum* value is shown.

Of course, the absolute values are an artifact of this system. However, it can be seen that any of the protocols can be tuned to optimize performance for mass data movement. The exception is Chirp, which only reaches about one half of the available bandwidth. This bandwidth limit is due to the strict RPC nature required for Unix semantics; the Chirp server does not extract from the underlying filesystem any more data than necessary to supply the immediate `read` operation. Although it is technically feasible for the server to read ahead in

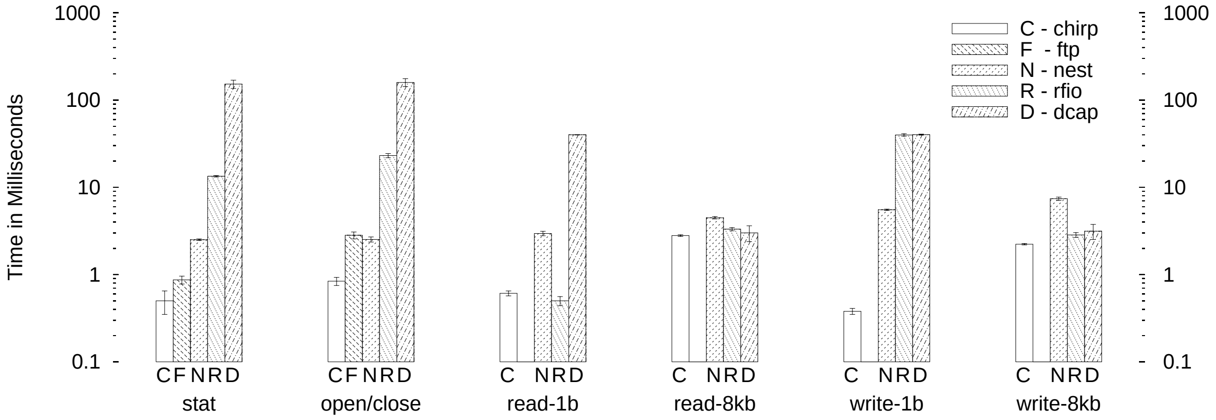


Figure 4.8: Latency by Protocol

The latency of Unix-equivalent operations by protocol. Note that this figure is log scale. The latency of each protocol differs by several orders of magnitude.

anticipation of the next operation, such data pulled into the server's address space might be invalidated by other actors on the file in the meantime and, strictly speaking, is semantically incorrect.

The hiccup in throughput of DCAP at a block size of 64KB is an unintended interaction with the default TCP buffer size of 64 KB. The developers of DCAP are aware of the artifact and recommend changing either the block size or the buffer size to avoid it. This is reasonable advice, given that all of the protocols require tuning of some kind.

Figure 4.8 shows the latency of Unix-equivalent operations in each I/O protocol. Each bar shows the mean and variance of 1000 cycles of 1000 measurements over a local area network with both the client and server on identical machines as described in Chapter 3. Take note that the graph is log scale: performance varies by several orders of magnitude.

I hasten to note that this comparison, in a certain sense, is not "fair." These data servers provide vastly different services, so the performance differences demonstrate the cost of the service, not necessarily the cleverness of the implementation. For example, Chirp and FTP achieve low latencies because they are lightweight translation layers over an ordinary file system. NeST has somewhat higher latency because it provides the abstraction of a virtual file system, user namespace, access control lists, and a storage allocation system, all built on an existing filesystem. The cost is due to the necessary metadata log that records all such activity that cannot be stored directly in the underlying file system. Both RFIO and DCAP are designed to interact with mass storage systems; single operations may result in gigabytes of activity within a disk cache, possibly moving files to or from tape. In that context, low latency is not a concern.

That said, several things may be observed from Figure 4.8. Although FTP has benefitted from years of optimizations, the cost of a `stat` is greater than that of Chirp because of the need for multiple round trips to fill in the necessary details. Likewise, the additional latency of `open/close` in FTP is due to the multiple round trips to establish a new TCP connection. FTP does not support fine-grained reads and writes, so no such measurements are shown. Both RFIO and DCAP have higher latencies for single byte reads and writes than for 8KB reads and writes. This latency is due to buffering which delays small operations in anticipation of further data. Most importantly, all of these remote operations exceed the latency of the debugger trap itself by several orders of magnitude, which validates the earlier decision to make use of the expensive reliability of the debugger trap.

Figure 4.9 shows the cumulative effects of these microbenchmarks on real applications. Each bar shows the mean and variance of ten runs of each application accessing all of its data over a local area network, using each of the various protocols. Note that this graph is *not* log scale.

Unfortunately, the RFIO and DCAP protocols were not able to successfully run any of these applications to completion. The RFIO protocol, because of its profligate use of TCP connections, quickly consumed kernel-level networking resources and caused both client and server to hang. The DCAP protocol does not include operations for manipulating directories, which each of these applications required to a limited extent. The figures for the protocols are

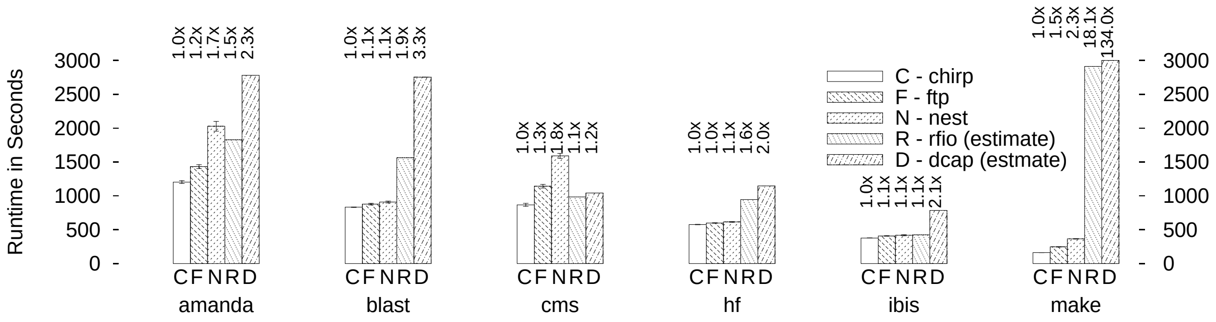


Figure 4.9: Application Performance by Protocol
The runtime of real applications accessing data over a wide area network by each protocol. The RFIO and DCAP figures are estimates based on the previous figure. Note that this figure has linear scale.

estimates computed from traces of each application's I/O activity and the microbenchmarks measured above.

As can be seen, the micro-design of an I/O protocol can have a significant effect on the overall performance of real applications. Because of the large number of metadata operations, the low latency provided by Chirp trumps any benefits in bandwidth available in the other protocols. This effect is extreme in the metadata-intensive `make` but also significant in the scientific applications. For example, `amanda` via FTP is is 1.2x slower than `amanda` via Chirp.

4.8 Conclusion

Several conclusions may be drawn from this chapter.

First, I have shown that any semantic transformation in data manifests itself as a performance penalty in the larger system. The Chirp protocol offers the best performance for these applications against standard I/O servers simply because it bears the closest resemblance to the needs of the job: that is, a single `open` in the job results in a single `open` in the Chirp server. In the other protocols, the need to perform multiple network operations for each I/O operation in the job – and multiple I/O operations in the server for each network operation received – adds up for real applications.

Second, a more positive interpretation of these results is to observe that protocol transformation *can be done*. It is quite feasible to dispatch an application into a distributed system, using an agent to attach to a variety of storage devices, each chosen by their owners for varying social and technical reasons. A diverse ecology of storage systems is no obstacle to the use of real applications.

Finally, protocol transformation is never leakproof. For each of the protocols used – including the most Unix-like, Chirp – there are error cases that do not translate into the interface expected by the job. These *escaping errors* will be the focus of the following chapter.

Chapter 5

Computation Coupling

5.1 Introduction

So far, I have exampined the details of coupling an agent to a variety of storage systems. Now, I will consider the relationship between an agent and the CPU management system that oversees it.

Traditionally, batch computing systems have provided jobs with a very simple interface. Once placed on a remote machine, a job is started via `fork` and `exec`. Small amounts of information may be passed to the job by way of environment variables. Once the job is complete, it calls `exit` to pass a single integer back to the CPU management system. No other information flows from the job back to the batch system.

This simple interface appears to be natural: after all, it matches the interaction between a parent and child process in a standalone operating system. However, it is not sufficiently rich to communicate the information needed for a data-intensive job, particularly one that is represented by an agent. There are two reasons for this. First, a job and its agent may require the help of the batch system in order to take advantage of the resources available to it. I will refer to any resource provided by a batch system to a running job as a *runtime service*. Second, the use of agents and runtime services increases the complexity of the system such that a more subtle interface to indicate the success or failure of a job is needed.

For example, one runtime service might provide a high-level execution environment for a particular language or system such as Java [17], MPI [153] or PVM [115]. The runtime service would be responsible for creating the execution environment and then placing the job in it. However, if this service should fail to work correctly, there must be some way of distinguishing between a service failure and a job failure: it is not the job's fault if Java was incorrectly installed!

A job may have arrived an an execution site via a variety of security mechanisms such as GSI [53] or Kerberos [137]. In order for the agent to operate correctly, a runtime service must direct it to local copies of its delegated credentials. However, these credentials can have unexpected properties: they may be time limited and expire while in use. This is also not the fault of the job itself; although, it might be the fault of the controlling user!

A runtime service might be located behind a firewall or some sort of network translation device. In order to allow an agent to communicate with the outside world, a runtime service must provide it with a network proxy such as SOCKS [85] or GCB [134]. These systems, too, can be the source of failures that should not reflect negatively upon the job itself.

Thus, runtime services are a two-edged sword. They can provide a job and agent with powerful capabilities to connect with external resources, but an extraordinary set of new failure modes that an agent and the surrounding system must come to grips with. How can we provide runtime services without making the system overly sensitive to new failures?

5.2 Case Study: The Java Universe

To make this discussion concrete, I will explain how I have added runtime services to the Condor distributed batch system in order to support Java applications. In order to describe these services in detail, I must make a slight detour to explain Condor itself.

5.2.1 Condor

The Condor distributed batch system creates a high-throughput computing system on a community of computers. A high-throughput system seeks to maximize the amount of computation done over a long period of time measured in months or years. (In contrast, a high-performance system seeks to maximize the computation performed in seconds of minutes.) A community of computers may be any configuration of machines that agree to work together, ranging from a single large SMP to a managed PC cluster to a set of workstations spread around the world. Condor was originally designed to manage jobs on idle cycles culled from a collection of personal workstations [90], and so is uniquely prepared to deal with an unfriendly execution environment by using tools such as process migration [133] and transparent remote I/O [92].

The core components of Condor are shown in Figure 5.1. Each participant of the system is represented by a daemon process that represents its interests. A user submits jobs to a *schedd*, which keeps the job state in persistent storage, and works to find places where the job may be executed. Each execution site is managed by a *startd* that enforces the machine owner's policy regarding when and how visiting jobs may be executed. The requests and requirements of both parties are expressed in a declarative language known as ClassAds [118], and forwarded to a central *matchmaker*. This process collects information about all participants, and notifies schedds and startds of compatible partners. Matched processes are individually responsible for communicating with each other and verifying that their needs are met. In this case, schedds and startds communicate directly to claim one another and verify that their requirements are met. Once matched, each creates a process to oversee the execution of one job. The schedd starts a *shadow*, which is responsible for providing the details of the job to be run, such as the executable, the input files, and the arguments. The startd creates a *starter*, which is responsible for the execution environment, such as creating

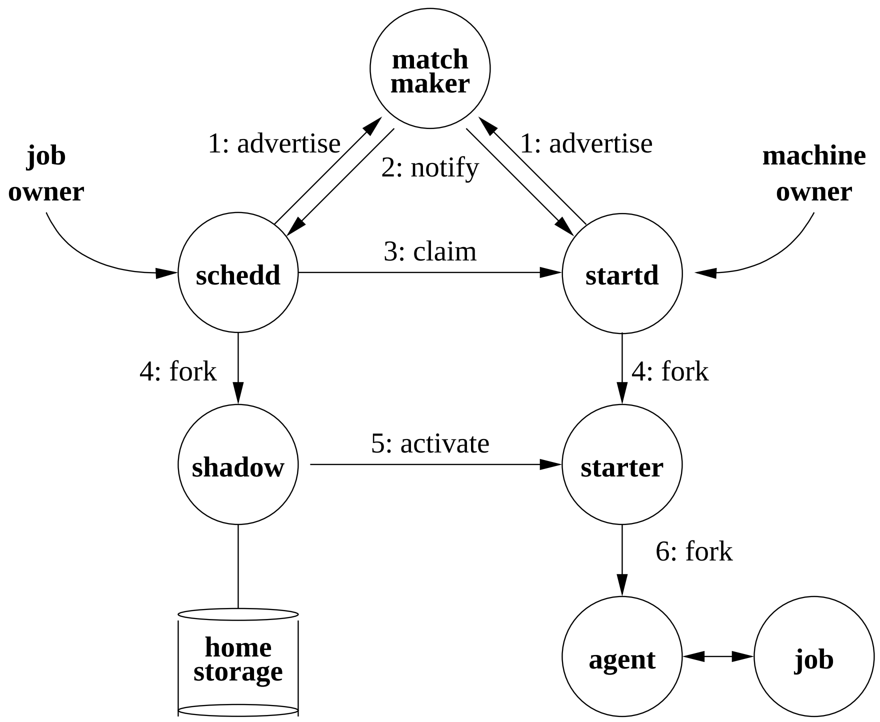


Figure 5.1: Overview of Condor

How Condor runs a single job. 1: The schedd and startd advertise themselves to the matchmaker. 2: The matchmaker informs the two processes that they are compatible. 3: The schedd claims the startd. 4: The schedd and startd fork a shadow and starter. 5: The shadow activates the starter. 6: The starter forks the agent and job.

a scratch directory, loading the executable, and moving input and output files.

Condor provides several *universes* for executing jobs. A universe is a carefully-selected set of features that create environments suitable for different kinds of jobs. The starter and shadow are responsible for working together to create a suitable universe. The Standard Universe provides transparent checkpointing and remote I/O capabilities for binary executables. It requires the program to be re-linked with a Condor-provided library. The Vanilla Universe can execute normal scripts and binaries that are not re-linked, but such jobs cannot checkpoint or migrate. Specialized universes are provided for the Globus [57], PVM [115], and MPI [153] environments.

5.2.2 The Java Universe

A growing number of number of users are turning to Java as a suitable language for distributed computing [56, 60]. Although Java code may not always execute as quickly as native machine code, it is believed that this loss is more than offset by faster development times and a larger pool of available CPUs.

To support this community, I have added a Java Universe to Condor. The added components are shown in Figure 5.2. In this universe, the starter does not execute the job directly, but instead invokes the JVM which in turn invokes the user's Java program. The JVM binary, libraries, and configuration files are all specified by the machine owner, as they are certain to differ from location to location. The user simply specifies the Java Universe, and does not need to know the local details.

The starter can transfer the job's input and output files at the beginning and end of execution. However, many jobs require more extensive I/O, perhaps from a selection of files that are impractical to transfer all together for every execution. For such programs, a simple I/O library is provided. This library presents files using standard Java abstractions,

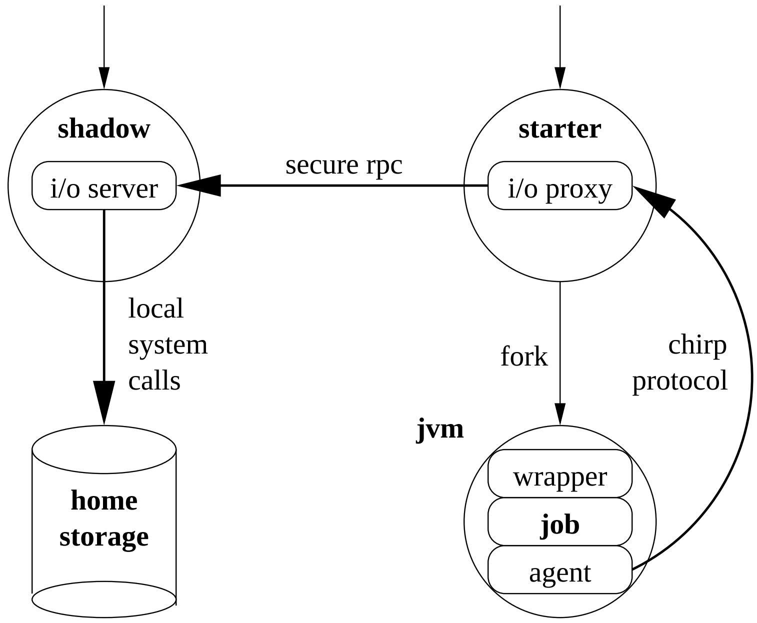


Figure 5.2: The Java Universe
This figure shows the components of the Java Universe within the Condor system. The starter invokes the JVM, which contains the wrapper (described in Section 5.4), the job itself and the agent library. The agent library performs I/O by communicating with a proxy in the starter. The proxy accesses the user's home storage by communicating with the shadow process.

such as the `InputStream` and `OutputStream` interfaces. This library is an agent based on polymorphic extension, as discussed in Chapter 3. The user must make small changes to the program in order to construct objects of type `ChirpInputStream` and `ChirpOutputStream`, but otherwise may perform I/O normally.

This library does not communicate directly with any storage resource, but instead interacts with a proxy in the starter via the Chirp protocol introduced in Chapter 4. The connection is established from one process to another on the loopback network interface. The library authenticates itself to the starter by presenting a shared secret revealed to it through the local file system. Thus, the connection is secure to the same degree as the local system.

The proxy allows the starter to transparently add additional functionality to the job without placing any burden on the user. The proxy allows the job to be transparently connected to runtime services that may vary from site to site or require special permissions. For example, a firewall could be crossed using configuration and authentication information that was only known by the proxy at the execution site. In this chapter, I will discuss a typical application of the proxy by making use of the standard Condor remote I/O channel to the shadow. This facility provides UNIX-like file access in the form of remote procedure calls secured by GSI [53] or Kerberos [137].

5.2.3 Initial Experience

Brave early users of the Java Universe were disappointed. Under ideal conditions, jobs would execute as expected. However, nearly any failure in a component of the system would cause the job to be returned to the user with an error message. If the Java installation was somehow faulty – the machine owner might give an incorrect path to the standard libraries – the job would exit and return to the user for consideration. If the job consumed more memory than was available on the machine, the job would exit, indicating an `OutOfMemoryError`. If the shadow's shared file system became temporarily unavailable, the job would exit indicating a `ConnectionTimedOutException`.

This behavior was correct in the sense that users received *true* information about how their jobs executed. However, it was undesirable because it required frequent postmortem analysis to determine whether the job had exited of its own account or simply because of accidental properties of the execution site.

I also found this frustrating, as I had gone to great trouble to assure that no error value was left unconsidered. For example, I ensured that file system errors discovered by the shadow were transmitted to the starter, and then converted into corresponding exceptions by the Java I/O library. At process completion, the exit code of the JVM was transmitted carefully back to the shadow, then the schedd and the user.

Fault-tolerance techniques such as replication and retry were not germane to this problem. Users wanted to see program generated errors such as an `ArrayIndexOutOfBoundsException`, but wanted to be shielded against incidental errors such as a `VirtualMachineError`. Knowledge of such details might be useful to users or administrators as a measure of system health, but were not useful as a program result.

5.3 Theory of Error Scope

To better understand this problem, we require a theory of error propagation. I will first describe key concepts relating to errors and then embark on a discussion yielding several succinct design principles. My goal is not to design new algorithms for fault-tolerant systems. Rather, I wish to bring some structure to the analysis of errors. Once an error is understood, then we may rewrite, retry, replicate, reset, or reboot as the condition warrants.

5.3.1 Terms

The generally accepted definitions of fault, error, and failure in computer science are those given by Avizienis and Laprie [18]. To paraphrase, a *fault* is a violation of a system's underlying assumptions. An *error* is an internal data state that reflects a fault. A *failure* is an externally-visible deviation from specifications.

For example, consider a machine designed to tally votes in an election and display the name of the candidate with the most votes. A fault might be a random cosmic ray that passes through the machine and corrupts some storage. If the corrupted storage contained program data in use, then the changed data would constitute an error. If the error was significant enough to alter the victor, then the machine would have experienced a failure.

A fault need not result in an error, nor an error in a failure. This may be through accident: The cosmic ray might corrupt storage not in use. Or, it may be through design: There may be multiple redundant machines that themselves must vote on the final output.

These three terms seem to work well when we consider a complex piece of machinery as a whole. However, they are not so useful when we begin to consider software as a collection of components. In a given piece of software, an error may or may not be a failure, depending on whether the software is the highest layer perceived by the user. Furthermore, the term *error* is often used to indicate a condition that is not desirable or not useful – i.e. "out of memory" – even though it may constitute a valid or expected state of the software.

In this context, it is more useful to use the generic term *error* to mean merely *an undesirable result* and then divide it into three categories that distinguish how an error makes itself known to the caller, whether it be a user or another piece of software.

An *implicit error* is a result that a routine presents as valid, but is otherwise determined to be false. For example, it would be an implicit error for $\sqrt{3}$ to evaluate to 2. It can be expensive to detect an implicit error, typically requiring duplication of all or part of a computation.

An *explicit error* is a result that describes an inability to carry out the requested action. For example, a routine to allocate memory may return a null pointer indicating "out of memory." Explicit errors require no further work to determine that they *are* errors, but may require additional work on the part of the caller to determine the next course of action.

An *escaping error* is a result accompanied by a change in control flow. This sort of error is not given directly to the caller of a routine, but to a higher level of software. An escaping error is necessary when a routine is unable to perform its action and is also unable to represent the error in the range of its results.

It is important to re-emphasize here that the terms *error* and *result* are semantically indistinguishable. A result that carries the message "out of memory" is simply a true statement. Whether it is desirable or not is a matter of opinion to the receiver of the message. (One could image a program that is *trying* to exhaust memory.) We simply have the convention of calling some results *errors* because they are undesirable to most programs. The distinction between the three kinds of errors lies in their method of propagation, not in the information that they carry.

Both explicit and escaping errors have been represented in recent languages by the *exception* [62]. The exception is a language feature that combines an object for carrying rich error information along with a change of control flow that allows the error to be propagated beyond the immediate caller. The exception is a useful programming tool, and I am generally in favor of its use to improve the readability and verifiability of programs. However, the use of exceptions is neither necessary nor sufficient for building a disciplined system. I will give examples that make use of exceptions, but offer discussion in terms that can be applied to any error representation, whether it be signals, strings, integers, or exceptions.

5.3.2 Error Relationships

To illustrate the relationship between the three error types, consider a standard virtual memory system that provides the illusion of a large memory space by making judicious use of limited physical memory and a larger backing store. Suppose that it discovers an explicit error: the backing store is damaged or unavailable. If it cannot satisfy an application's `load` operation, what should it do? A `load` operation has no result that can signify an error.

The system might return a random or default value, thus creating an implicit error in the calling layer. This would clearly be an unacceptable design. Implicit errors are difficult enough to detect when they are introduced through accident or physical faults. We must not add to the problem by making them a deliberate presence.

Principle 1 *A program must not generate an implicit error as a result of receiving an explicit error.*

The system may attempt to apply any number of standard techniques in fault tolerance. It may consult mirrored copies or simply retry the operation. But what if these fail or timeout? The system must cause an escaping error rather than corrupt the results with an implicit error.

The escaping error is not simply the crutch of a novice programmer that lazily calls `abort` rather than handle an uncomfortable boundary condition. It is a vital component of a system programmer's toolbox that must be used when a routine is in danger of violating an interface specification. The escaping error is a disciplined exit resulting in an explicit error at a higher level of abstraction. It can be communicated in a variety of ways, depending on the form of the communication interface. On a network connection, an escaping error is communicated by breaking the connection. Within a running program, an escaping error is communicated by stopping the program with a unique exit code. In this case, a virtual memory system communicates an escaping error by forcibly killing the client process, which then exits with a signal indicating a memory error.

Principle 2 *An escaping error must be used to convert a potential implicit error into an explicit error at a higher level.*

The need for the escaping error is obvious in an interface that cannot express errors, such as the virtual memory system mentioned above. Yet, it is still necessary in interfaces that express explicit errors. Consider this interface used to access a file:

```
int open( String filename ) throws FileNotFound, AccessDenied;
```

The exceptions `FileNotFound` and `AccessDenied` are explicit errors that describe an inability to carry out the caller's intentions. However, these explicit errors are ordinary results in the sense that they conform to the function's interface. A well-formed caller of `open` must be prepared to deal with these eventualities in some way. Indeed, one purpose of the exception mechanism is to ensure that the caller deals with all expected results. The appearance of these errors does not violate the contract of the function in any way.

However, no interface can capture all of the possible implementation errors of a routine. Every routine rests on many unstated assumptions such as the coherency of memory and the infallibility of a function call. Such violations, even if detected, are generally considered beyond the concern of the designer.

Thus, the escaping error represents the mismatch between an interface and an implementation. A new file system may be built in terms of disk operations, network communications, carrier pigeons, or other mechanisms not yet imagined. In order to attach such systems to existing interfaces, we must deal with error values that do not fit into an existing interface. Regardless of the interface, a function such as `open` may be susceptible to a `PigeonLost` if it is given an avian implementation [151]. An escaping error is a symptom of a fundamental incompatibility between two systems.

5.3.3 Error Scope

To be accepted by end users, a system must be sensitive to the distinction between the explicit and the escaping error. If the system can successfully create the computation environment expected by the user, then a program's result, error or otherwise, must be returned to the caller. If the system is unable to create the expected environment, then an escaping error distinguishable from a program result must be delivered to the surrounding system.

However, the use of the escaping error raises a conundrum. In order to accept and react to an escaping error, a system must be able to understand its meaning to a certain degree. But, the very nature of an escaping error is to describe an implementation dependent problem that may vary from system to system. To solve this problem, we need an abstraction that balances the diagnostic need for information with the principle of separation between implementation and interface.

I introduce the abstraction of error scope to solve this problem. The *scope* of an error is simply the portion of a system which it invalidates.

For example, `FileNotFound` has file scope. It simply states that the named file cannot be found. A failure in remote procedure call (RPC) [29] has process scope. It indicates that the mechanism of function call is no longer valid within the process. A node failure in PVM [115] has cluster scope. If one node crashes, then the whole cluster of nodes is obliged to fail.

In each case, an error must be interpreted by the program (or process, routine, function, etc.) that is responsible for managing that error's scope. For example, the calling function is capable of handling an error of function scope. The creator of a process is capable of handling an RPC error of process scope. The creator of a PVM cluster is capable of handling an error of cluster scope.

Principle 3 *An error must be propagated to the program that manages its scope.*

An error's scope may be re-considered at many layers. It may gain significance, or expand its scope, as it travels up through layers of software. For example, at the level of network communications, an error indicating a lost connection is simply that. However, when interpreted in the context of RPC or PVM, it becomes an error of process or cluster scope, respectively.

In many cases, there may be a specialized mechanism for delivering an error to the manager of its scope. For example, a POSIX signal can deliver an error directly to a parent process. In other cases, we may use an indirect channel, such as a file, to carry the necessary information to its destination. We will see an example of this below.

5.3.4 Generic Errors

A frequent source of confusion in error propagation is the generic error. A *generic error* is an indication that a routine may return any member of an expandable set of related errors. Such an interface makes a very weak statement about the behavior of a routine, creating confusion for both the implementor and the caller.

An example of a generic error may be found in the Java I/O system. Consider this interface fragment:

```
class FileWriter {
    FileWriter( File f ) throws IOException;
    void write( int )    throws IOException;
}
```

The generic error `IOException`, thrown by both both methods, is defined by the standard Java package and is extended to include a variety of exception types such as `FileNotFound` and `EndOfFile`. Users of these interfaces are encouraged to create new error types that extend the basic type. This appears attractive: flexibility and generality are usually seen as programming virtues. However, this generic interface creates problems with both the errors it includes and those it omits.

The use of `IOException` suggests that both methods are subject to the same set of explicit errors. This is certainly not the case in most I/O systems. Traditionally, the act of opening a file is subject to errors of permission and existence that occur while navigating a namespace. Once opened, the file is locked in such a way that reads and writes are sure to succeed, subject to the bounds of the file size. Would it be reasonable for an implementation of `write` to throw a `FileNotFound`? Of course not! That would violate the ordinary expectations we have of an I/O system. Even if we could manage to build a bizarre distributed file system subject to losing a file in the middle of a `write`, we would expect to receive an escaping error, not an explicit error. We know this only because we are familiar with the conventions of I/O systems. If we were to encounter a generic error in a less familiar interface, the behavior would not be so obvious.

Despite the invitation to extension, there is little practical way to make use of an error type not mentioned in the originally documented instances of `IOException`. Suppose that we wish to know when the file system runs out of space. (This possibility is not mentioned in the Java documentation.) From the caller's perspective, we have no idea how an implementation will behave. Guessing at exception names is futile: the names `DiskFull` and `FullDisk`, as well as many others, are plausible names. From the implementor's perspective, we have no idea if the caller is prepared to deal with this error. Can it handle an `DiskFull` or would it be better to retry and hide the error? At least one Java implementation avoids this problem entirely by blocking indefinitely when the disk is full. The generic error offers us no help in deciding whether other implementations will behave this way.

If we wish to make a caller and an implementor agree on a convention for a `DiskFull` error, we must establish some way for them to know that the other is aware of the convention. To know this would violate the principle of separation between interface and implementation, unless we simply create a new interface that describes `DiskFull`.

I conclude that the generic error leads to more questions than answers. Rather than bringing structure to an interface, it forces the participants to make guesses. I advocate that an error interface is only useful when it makes a strong, limited statement. It is better to exclude a `DiskFull` error entirely then to leave the participants guessing at its existence.

Principle 4 *Error interfaces must be concise and finite.*

If it was possible to revise these I/O interfaces to conform to Principle 4, I would write something like this:

```
class FileWriter {
    FileWriter( File f ) throws FileNotFound, AccessDenied;
    void write( int )    throws DiskFull;
}
```

If this revised interface were to be used in a context with the possibility of a new type of fault, such as `ConnectionLost`, then it must be communicated with an escaping error according to Principle 2. If the caller wishes to deal with such an error explicitly, then a new interface must be constructed to inform both parties of their mutual interest.

5.4 Java Revisited

To apply these ideas to the Java Universe, we must first identify the system's various error scopes and their handling programs, shown in Figure 5.3. Dotted lines indicate scopes, circles indicate handling programs, squares indicate resources, and arrows indicate return values.

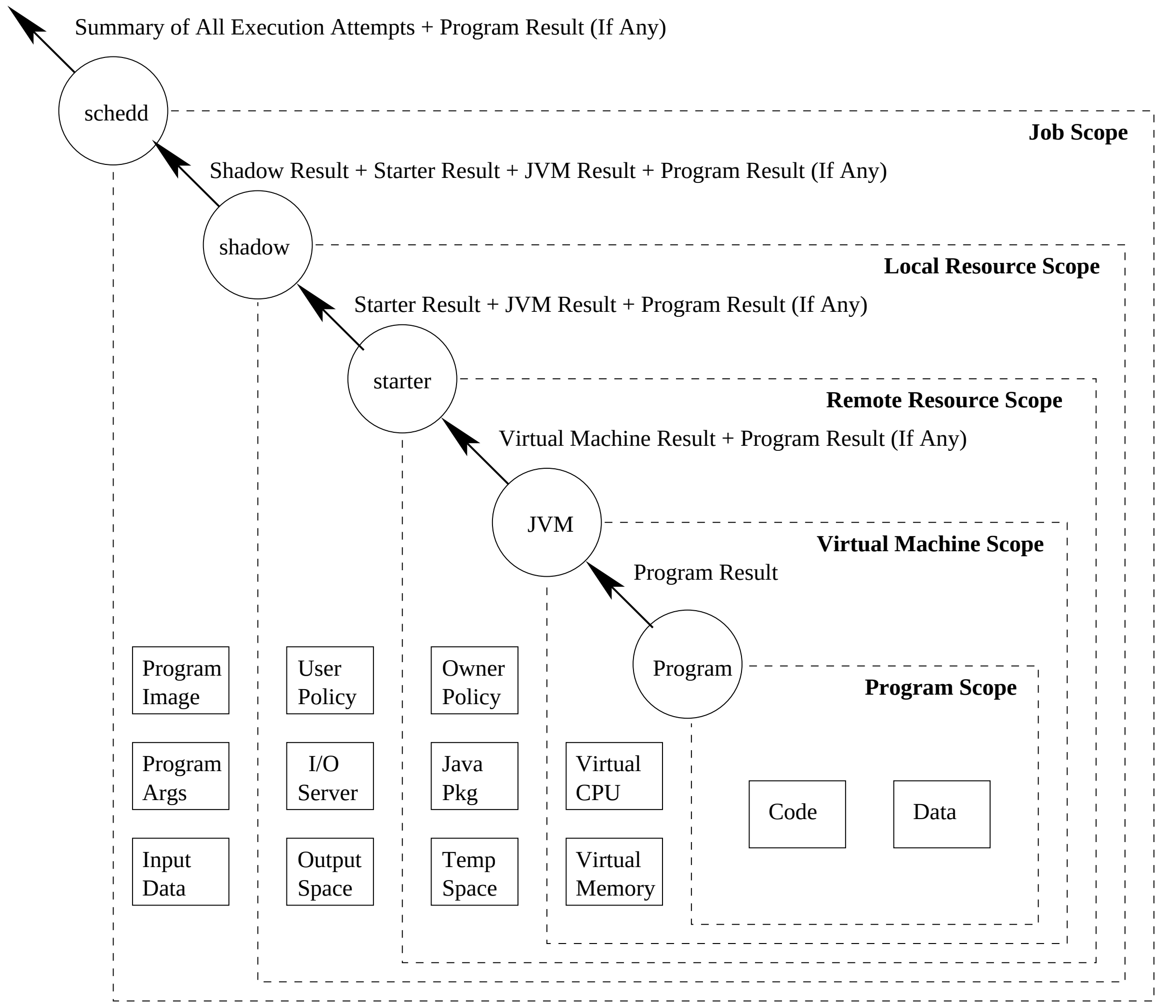


Figure 5.3: Error Scopes in the Java Universe

This figure shows the error scopes and handling programs in the Java Universe. Each dotted rectangle indicates a scope. Attached circles indicate the handling program for each scope. Squares indicate resources that are members of each scope. Labeled arrows indicate return values from one handler to another.

Execution Detail	Error Scope	JVM Result
The program exited by completing `main`.	Program	0
The program exited by calling `System.exit(x)`	Program	x
Exception: The program de-referenced a null pointer.	Program	1
Exception: There was not enough memory for the program.	Virtual Machine	1
Exception: The Java installation is misconfigured.	Remote Resource	1
Exception: The home file system was offline.	Local Resource	1
Exception: The program image was corrupt.	Job	1

Figure 5.4: JVM Result Codes

This figure shows the result code generated by the JVM for various possible program executions. The result code is not useful, because it does not distinguish error scopes. A result of 1 could indicate a normal program exit, an exit with an exception, or an error in the surrounding environment. This problem is solved by the wrapper described below.

Each process in the system is responsible for managing certain physical resources. Error scopes correspond directly to management responsibility. For example, a corrupted program or a missing input file has job scope. In such a case, the schedd is responsible for informing the user that the job cannot run. An unavailable file system has local resource scope. The shadow would be responsible for informing the schedd that the job cannot run right now. In contrast, a misconfigured JVM has remote resource scope. The starter would be responsible for informing the shadow that the job cannot run on the given host. A lack of memory for the program has virtual machine scope. The JVM would be responsible for informing the starter that the job cannot run in the current conditions.

In each scope, the managing program could apply fault-tolerance techniques to mask the error, or it could propagate the error up the chain. If it chooses the latter, it must distinguish between errors in its own scope and errors in containing scopes. The last line of defense is the schedd. If it detects an error of program scope, it identifies the job as complete and returns it to the user. If it detects an error of job scope, it identifies the job as unexecutable and also returns it to the user. Anything in between causes it to log the error and then attempt to execute the program at a new site.

With this understanding, the problem with the Java Universe becomes clear: I failed to apply Principle 3 and direct errors to the manager of each scope. Several small changes throughout the system were necessary to fix this problem. Here are two examples.

While executing a Java program, the starter relied entirely on the exit code of the JVM as an indicator of program success. This simple (but incorrect) assumption introduced implicit errors, violating Principle 1. As Figure 5.4 shows, the JVM can cause an environmental error to appear as a program result. To retrieve the necessary information, it was necessary to add the *wrapper* shown in Figure 5.2. The starter causes the JVM to invoke the wrapper with the actual program as an argument. The wrapper locates the program, attempts to execute it, and catches any exceptions it may throw. It examines the exception type, and then produces a result file describing the program result and the scope of any errors discovered. The starter examines this result file and ignores the JVM result entirely.

While performing I/O, the agent blindly converted all possible explicit errors from the proxy directly into corresponding Java exceptions. For failure modes not represented by existing exception types, the agent simply extended the basic `IOException` to a new type. Although this was easy, it was incorrect. I gave in to the temptation offered by the generic error interface proscribed by Principle 4. Although errors such as "connection timed out" and "credentials expired" could technically be represented by an `IOException`, they violated a program's reasonable expectations of the I/O interface and thus fell outside of the program's scope. To propagate such errors correctly, I applied Principle 2 and modified the agent library to send an escaping error (a Java `Error`) to the program wrapper, which communicates the error scope to the starter through the result file.

The reader may fairly object that the "reasonable expectations" test is extraordinarily vague. I admit that there is room for disagreement in such a subjective term. It is precisely this confusion which motivates the statement of Principle 4.

With the changes described above, the hailstorm of error messages abated, and the system settled into a production mode.

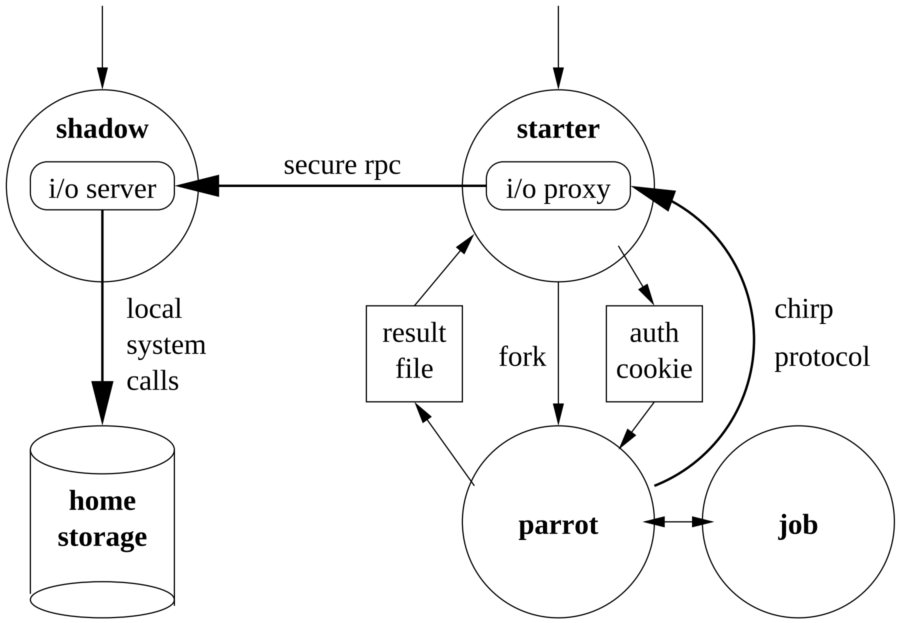


Figure 5.5: Parrot and Condor

Parrot interfaces with Condor in a manner similar to that of the Java Universe. Parrot traps the job's system calls and then may perform I/O via the Chirp protocol to the I/O proxy. It may also perform I/O to other services, using the Chirp proxy only for name resolution.

5.5 Parrot Revisited

The use of runtime services and the concept of error scope maps well to other jobs and agents. For example, Parrot can be used to couple ordinary Unix programs to the same Chirp proxy, as shown in Figure 5.5. Just as in the Java Universe, files must be used to communicate the authentication cookie as well as the result file between the starter and Parrot. The error scopes shown in Figure 5.3 apply equally well, by simply substituting Parrot for the JVM. Likewise, the problem of error codes shown in Figure 5.4 is Parrot's problem as well: the job's result and the scope of any errors must be communicated in the result file.

The previous chapter described many ways in which Parrot can emit an escaping error. In this context, the escaping error is simply recorded in the result file, describing the scope of the error for the benefit of the system and the details of the error for the benefit of the debugging log. Parrot may then exit, allowing the starter to propagate the error appropriately.

An agent such as Parrot may also be used to create a cross-coupling between CPU and I/O scheduling when data services may be the source of a delay. The storage services that are available for use now are relatively passive devices. However, there are a number of services in development that will be responsible for moving files over the wide area in response to user requests. For example, systems such as SRB [23] and SRM [130] will present a filesystem interface to demand-driven data movement. FTP [112] is a popular front-end interface to hierarchical storage managers. Stork [80] serves as a reliable queue for data transfer requests. With all of these services, a request for data may result in a response indicating that the data are in delivery and will arrive in a given amount of time or even at a different location.

Using the Chirp interface, an agent can cause a running job to be rescheduled at a later time or a different place. The Chirp interface provides a `constrain` call that specifies a clause to be added to the job's scheduling constraints expressed in the ClassAd [118] language. For example, if the agent discovers that a needed file is not scheduled to be staged in from tape until 9:15 AM, it can use `constrain` to add the requirement `(DayTime() > '9:15:00')` and then exit, indicating an error of storage scope. Condor will not re-place the process until the new requirements are satisfied.

For another example, suppose that the agent has already moved a dataset to a nearby cache at considerable cost. Without releasing the CPU, it might call `constrain` to add the requirement `(Subnet=="128.105.175")` and then continue processing. If the application should fail or be evicted at a later time, it will be re-placed by Condor at any machine satisfying the constraints: that is, on the same subnet as the needed data. Multiple calls to `constrain` overwrite the previous constraint, so that previous decisions may be un-done.

The notion of job-directed resource management is introduced in J. Pruyne's doctoral thesis [114]. A resource management interface called CARMI permits a running job to request and release external resources at run-time. A similar idea is found in the notion of execution domains [24], where the Condor shadow directs future allocations based on the location of checkpoint images. The Chirp `constrain` facility combines both of these ideas by permitting the agent to direct further allocation requests on the job's behalf in response to the developing state of the system.

5.6 Conclusion

An agent is charged with the deceptively simple problem of coupling a job to a batch system. When a program stops execution, the agent must simply communicate whether the job succeeded, returning a result, or failed to complete for some external reason. As I have shown in this chapter, the usability of a system hinges upon making this subtle decision correctly. The notion of error scope allows us to structure error propagation, even if the evolution of a system creates new failure modes.

It is useful to think of the notion of error scope in conjunction with a virtual machine model. Each error scope shown in Figure 5.3 can be thought of as corresponding a virtual machine with specific resources to manage and a concrete instruction set or interface to implement. With a clear definition of the expectations involved at each level, it is easy to determine whether a particular error condition is compatible with or breaks the virtual machine model at a particular level.

Now, it is *possible* to build a system without these distinct boundaries. But, if no expectations are made or enforced at any level, then any exception is, in some sense, *valid* in any context. The result is that exceptions of previously-unknown types percolate up to the top level for consideration by the user. Unfortunately, distributed systems are very good at creating new types of errors, thus making users very unhappy.

The discipline of error scope is key to building reliable systems. We will see how error scope can be applied to a complete system in the next chapter.

Chapter 6

Coordinating Computation and Data

6.1 Introduction

So far, I have presented the a job's agent by detailing its coupling to three independent entities: the job itself, the storage system, and the computation system. I will conclude by describing the interaction of all of these components in a complete system for executing data intensive workloads.

The execution of a job may be thought of as a *transaction* [63] involving each of these resources. The inputs to the transaction are the executable code, the input parameters, and the input data files. The outputs of the transaction are the exit status of the job and the output files it has written. Within certain constraints given by the user, the data access and computation must be performed as a coherent whole. Transactions are traditionally described in the context of database systems that guard repositories of highly structured data with stringent consistency requirements. The canonical properties of a transaction are described as ACID: atomicity, consistency, isolation, and durability.

Atomicity requires that either all or none of a transaction's outputs are committed to the database. Consistency requires that an entire transaction must respect constraints promulgated by the administrator. Isolation requires that several concurrent transactions run as if only one was running. Durability requires that the transaction only indicate success if the outputs are on persistent storage.

A transaction in a batch setting needs only two of these properties: consistency and durability. I will refer to this type of limited transaction as a *CD-transaction*, which can either be thought of as consistency-durability or computation-data.

It is acceptable, perhaps even desirable, for the output of a batch job to be non-atomic. Because batch jobs may run for hours or days, the developing output can assure the user that the system is making progress, reveal problems in the code or the input parameters, and allow the cost of output to be amortized over the lifetime of a job. Batch systems do not need to enforce isolation between transactions because batch workloads generally have isolation encoded in their structures. For example, a large workload composed of simulations would have a set of jobs each reading distinct input parameters and writing to clean output files. Further, the assumption of isolation is critical to performance: if a batch system was required to enforce isolation, a large amount of communication would be required between otherwise independent execution nodes.

A CD-transaction must be consistent. That is, the results – the exit code and the output files – must be the result of the same computation. If it is necessary to repeat the same computation multiple times before success is achieved, the outputs must both be the results of the final attempt. The transaction cannot be considered complete until both results are received. A CD-transaction must also be durable. Throughout the lifetime of a transaction, the various components may be stored in volatile memory on a variety of machines, and perhaps even in durable storage on incidental devices such as remote disks. However, the transaction cannot be considered complete until all results are committed to stable storage in the location specified by the user.

Incidentally, the notion of a CD-transaction captures very nicely the semantic distinctions between batch and interactive data access. Lu and Satyanarayanan have made the case that interactive users on mobile computers require transactions that provide only isolation and

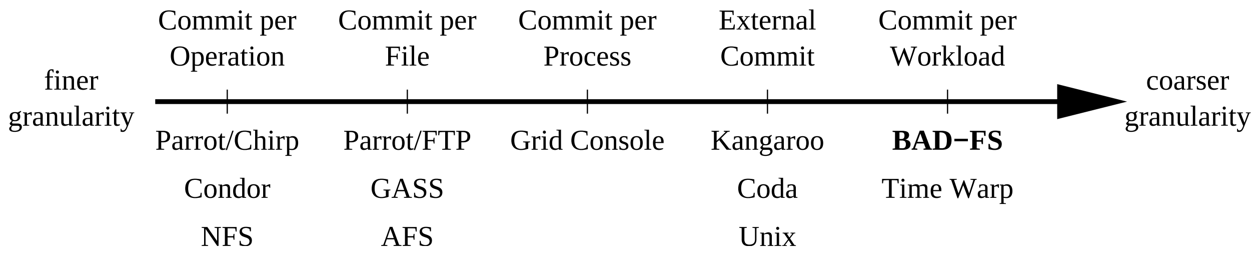


Figure 6.1: Transaction Granularity

no other property [94]. The assumption is that interactive users must be protected from meddling by other users, but otherwise will closely supervise and recover from failures. In a batch environment, isolation is not needed, but consistency and durability are, precisely because the user is not willing to supervise closely.

6.2 Transaction Granularity

A CD-transaction can only commit if both of its components – computation and data – successfully complete. That is, if a job runs successfully but its output data is lost in transit, then the transaction must not commit. Alternately, if a job successfully writes some output data, but the computation crashes or is otherwise lost, then the transaction must not commit. Of course, because a CD-transaction is not atomic, a non-committed transaction may leave behind some evidence of its execution, but the *job* cannot be recorded as complete unless the CD-transaction commits.

A CD-transaction is built up from many small components. A process writes its output by issuing filesystem commands such as `open`, `write`, and `close`. The data may be committed one operation at a time, in small groups, or all at the end of a job or workload. Each of these steps might be considered a *nested transaction* [100] or a *savepoint* [63]. This choice of transaction size is known as *transaction granularity*, and can be controlled by the agent on behalf of the job.

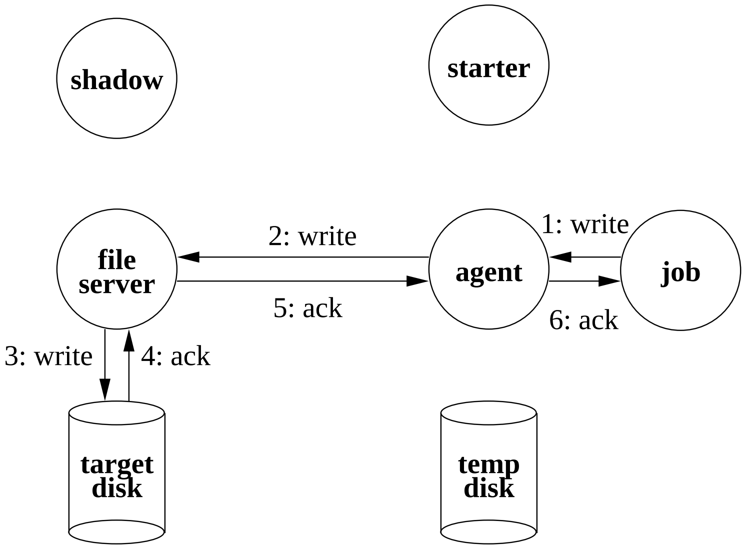


Figure 6.2: Commit per Operation
This figure shows a transaction granularity of commit-per-operation. Each write performed by the job results in a chain of messages all the way to the storage device, and then a chain of acknowledgements all the way back to the job.

Figure 6.1 shows a range of choices in transaction granularity along with the names of systems employing that granularity. When a fine commit granularity is used, a computation stays closely in sync with the outputs that it generates. A system with a fine commit granularity is easy to build, but is typically very sensitive to the availability and reliability of the communication and storage devices in use. As the commit granularity becomes coarser, a computation may become more and more out of sync with its output. A large commit granularity reduces the sensitivity of a system to the undesirable physical properties of a system, but requires increasing complexity to reconcile the computation and data portions of a transaction.

The simplest and finest granularity is *commit per operation*, shown in Figure 6.2. In this model, a job stays in close contact with its target storage. When the job requires input or output, the agent issues a remote procedure call to access some limited portion of the data that it needs immediately. A write operation must be immediately forced to stable storage before an acknowledgment may be given. Commit-per-operation is used by Parrot through Chapter 4 with partial file protocols such as Chirp. A small commit granularity is found many other places such as in the Condor remote I/O system [133], and in the strict interpretation of the NFS [126] distributed file system protocol. (However, most users of NFS do not actually see this behavior, because the strict protocol is typically hidden behind a buffer cache.)

Commit-per-operation is easily constructed and well-understood. However, it has two major drawbacks. The first is that the job's I/O performance is latency-bound: Every I/O operation must suffer the round trip latency of the network and the storage device. We saw this dramatic effect earlier on metadata-intensive programs such as `make`. The second problem is that commit-per-operation holds the job hostage to the whims of the larger system. If the storage device should crash, or the network fail, or a temporary load diminish bandwidth, the job will pause as it waits for the remote procedure call to finish. This pause will occur regardless of whether the I/O is performed over a reliable connection, as in Chirp, or over connectionless datagrams, as in NFS.

We may address these problems to a certain extent by changing the transaction granularity to *commit-per-file*. Whenever a file is opened, it is copied whole from remote storage and placed in local storage where the job may access it at local speeds. When the file is closed, if changed, it is written back to the target storage device. This is the approach used by Parrot with whole-file protocols such as FTP. It is also found in the GASS [28] system as well as the AFS [68] distributed file system.

This approach improves the latency of individual reads and writes at the expense of fetching the whole file at first open. Commit-per-file has been shown [68] to improve scalability for traditional interactive workloads. However, it requires the attention of the job when a file is closed. In Unix, a close operation cannot fail; it is simply a release of resources. When using commit-per-file, the closing of a file can fail; few applications are prepared to deal with

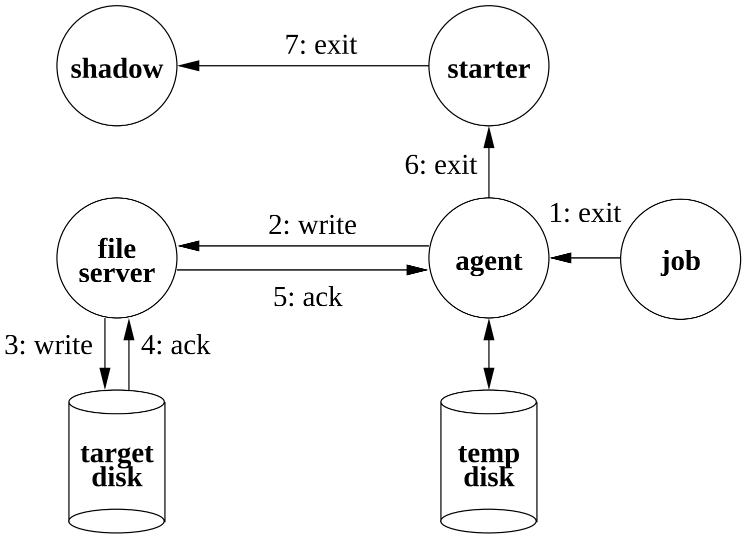


Figure 6.3: Commit per Process
This figure shows a transaction granularity of commit-per-process. While it runs, a job may write data to a temporary disk. When it exits, the agent is responsible for pushing all data to the target disk before indicating an exit to the batch system.

such a failure. In the batch context, we can rely on the agent to detect a close failure and then emit an escaping error.

At even coarser granularity, we may perform *commit-per-process*, shown in Figure 6.3. As the job runs, the agent attempts to write its outputs to the remote storage device. If that is not possible, it simply buffers them in temporary storage. When the job exits, the agent must be sure to flush all buffered data to the file server before permitting the job to exit. If the outputs cannot be written, then the job exit fails, and the transaction remains uncommitted.

This technique is used in a tool called the Grid Console [141]. The Grid Console allows users to have an interactive-when-possible I/O service. When the network is available, outputs are immediately visible. When the network is not available, outputs are delayed, but jobs do not stop executing. Commit-per-process is valuable to users, because it preserves much of the human satisfaction of commit-per-operation without the attendant fragility.

However, it still remains that jobs will fail or be delayed by a network outage or slowdown.

A common (but dangerous) solution to this problem is to make use of *external-commit*, shown in Figure 6.4. In this model, a job writes its outputs to a nearby buffer. The buffer accepts data as quickly as possible while simultaneously moving data out to its eventual destination. When the job wishes to exit, the agent sends a **commit** operation to the buffer to ensure that all data are safe. If successful, the job may exit, but it is not complete yet, because the output has not been delivered. An external entity is needed to remember and commit the incomplete transaction. Once the job and the agent have left the execution site, the CPU may be used by other jobs while the buffer trickles data back to persistent storage. Finally, the external entity must issue a second operation called **push** or **sync** to ensure that the buffer's work is done.

This technique has been proposed in several settings. The Kangaroo [142] distributed buffer cache is a peer-to-peer network of identical servers, each providing buffer space for batch jobs. Kangaroo provides both the **commit** and **push** operations: the former makes data more resilient to crashes, while the latter ensures that it is committed to the target storage.

Anderson et al. [15] propose a similar concept whereby *buffer-servers* spread across a distributed system accept pending data and forward it to their ultimate destination. As proposed, buffer servers have a **commit** but not a **push**.

The ordinary semantics of a Unix system with a buffer cache may be seen as an example of external commit. In ordinary operations, applications write data into the buffer cache, and receive an acknowledgment that the cache has accepted the data. Independently of the application, the user or the kernel must periodically issue a **sync** operation in order to force all data to persistent storage.

The Coda [78] distributed file system has a similar concept, albeit with the opposite flow of information. Designed for partially-connected computers such as laptops, Coda allows

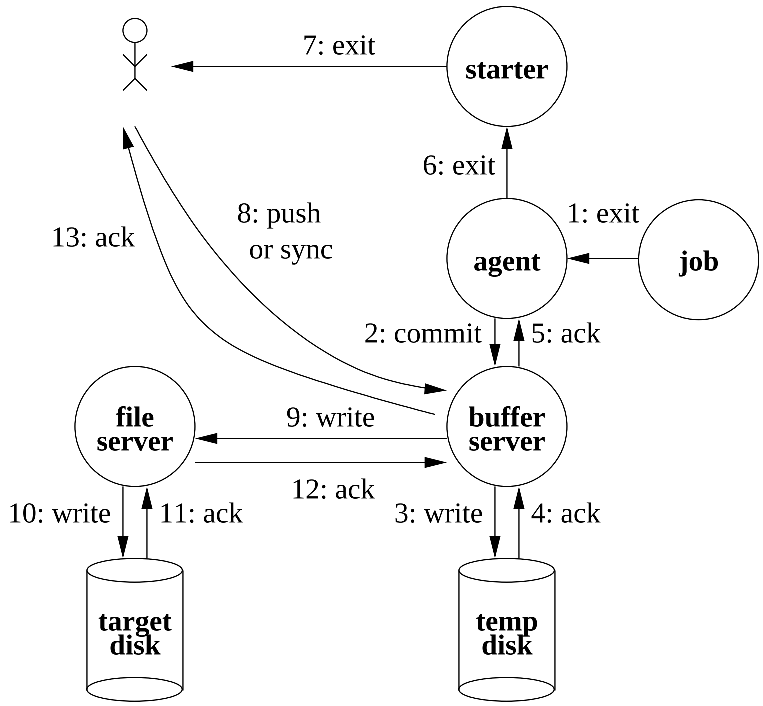


Figure 6.4: External Commit

This figure shows an externally committed transaction. While it runs, a job passes its data to a nearby buffer server. When it exits, the agent obtains an acknowledgement from the buffer server before indicating exit to the batch system. At a later time, the user externally probes the buffer server and obtains an acknowledgement when data movement is complete.

applications to write at will to a disconnected buffer cache. When the buffer cache is reconnected to the target storage, the two are reconciled, and any errors in the committal of data – such as non-isolated writes – are reported to the user by email.

External commit is useful in interactive computing systems because the initial writer sees a low latency. If an external commit should fail, then it is assumed the user is readily available to examine and recover the system. This approach is acceptable because the interactive user has intimate knowledge of the history and purpose of important files. Such an approach is not suitable in a batch environment because a batch workload may involve thousands of active files and processes. Without an automatic coupling between buffered data and the process that created it, the system cannot recover from commit failures automatically.

A more appropriate method for batch computing is to expand the notion of a transaction to the size of an entire batch workload. An entire workload may have several input files and several output files that are of interest to the user. Between the inputs and outputs may be a large number of jobs that use files merely as temporary storage. So long as the final outputs of interest are committed to stable storage, it may not be necessary to store or even transfer intermediate results.

A highly idealized implementation of commit-per-workload is found in the Time Warp [73] simulation model. In this model, parallel programs are written in a specialized message passing language. Each node maintains a virtual time counter in a manner similar to that of the Lamport clock algorithm [82]. If a node discovers that its virtual clock has run ahead of its peers, then a *causality error* has occurred. It may be the result of an over-optimistic speculation, or it may simply be the result of a failed node that has restarted. A causality error requires a node to retreat to a previously recorded checkpoint and send compensating messages to its peers to cause them to retreat as well. (The ideal selection of checkpoint times in a Time Warp system has been a matter of much debate [113, 104, 116].) In any case, the entire workload does not commit until all nodes have run to completion and written

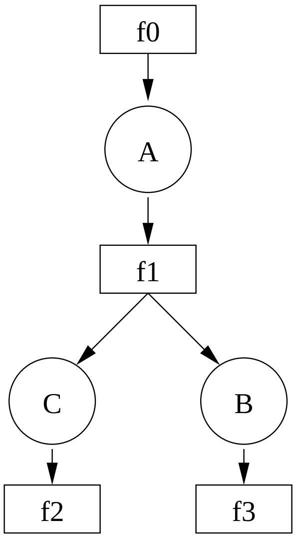


Figure 6.5: Commit per Workload
This figure shows a complete workload that may be treated as a transaction. Circles indicate processes to be run. Boxes indicate files read or written by each process.

their outputs to external storage.

We need not make the complete generalization of Time Warp in order to gain the benefits of the commit-per-workload model. We may consider Unix processes to be exchanging messages in the form of the files that they read and write. For example, Figure 6.5 shows a simple workload that may be considered as a transaction. The circles represent processes to be run, and the squares represent files. File `f0` is an input file, and files `f2` and `f3` are output files. The outputs are generated by running job `A` to produce `f1`, followed by jobs `B` and `C`.

By considering the entire workload as a transaction with intermediate files as distributed checkpoints, we are able to both improve performance as well as recover from a large number of failures. By declining to commit `f1`, we may improve the execution time of `A`. If job `B` fails to complete, it may be run again using `f1` as input, if it can be found. If not, then we know exactly how to regenerate it: by re-running `A`.

The commit-per-workload model allows us to take a holistic view of the computation and

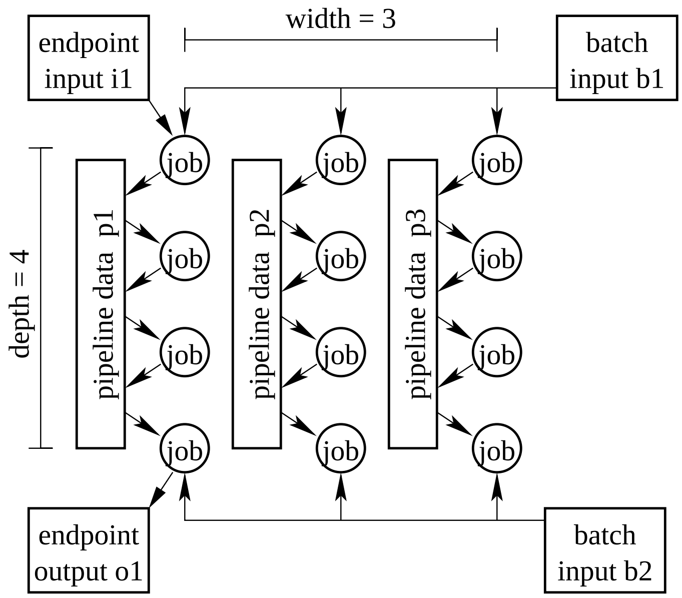


Figure 6.6: A Batch-Pipelined Workload

data needs of a large amount of work. I explore this approach next in a system called the Batch Aware Distributed File System, or BAD-FS.

6.3 Case Study: BAD-FS

The Batch Aware Distributed File System (BAD-FS) is a distributed system designed to run unmodified data-intensive batch applications on large-scale, unreliable computing systems. It relies on a commit-per-workload model in order to make its operation both fault-tolerant and performant.

BAD-FS is designed to run the type of scientific batch workloads introduced in Chapter 1. As illustrated in Figure 6.6, these data-intensive workloads are composed of multiple independent vertical sequences of processes that communicate with their ancestors and relatives via private data files. A workload generally consists of a large number of these sequences that are incidentally synchronized at the beginning, but are logically distinct and may correctly execute at a different rate than their siblings. These are known as *batch-pipelined* workloads.

One of the key differences between a single application and a batch-pipelined workload is file sharing behavior. For example, when many instances of the same pipeline are run, the same executable and potentially many of the same input files are used. One may characterize the sharing that occurs in these batch-pipelined workloads by breaking I/O activity into three types (as shown in Figure 6.6): *endpoint*, the unique input and final output; *pipeline-shared*, shared write-then-read data within a single pipeline; and *batch-shared*, input data shared across multiple pipelines.

The typical computing platform for batch-pipelined workloads is one or more clusters of managed machines spread across the wide area. I assume that each cluster machine has processing, memory, and local disk space available for remote users, and that each cluster exports its resources via a CPU sharing system. The obvious bottleneck of such a system is the wide-area connection, which must be managed carefully to ensure high performance. For simplicity, I will focus on the case of a single cluster being accessed by a remote user, but I'll conclude with an example of BAD-FS used across multiple clusters.

This organized and well-managed collection of clusters is known as a *cluster-to-cluster* (or c2c) system, in contrast to popular peer-to-peer (p2p) systems. Although the p2p environment is appropriate for many uses, there is likely to be a more organized effort to share computing resources within corporations or other organizations. Cluster-to-cluster environments are more stable, more powerful, and more trustworthy. That said, p2p technologies and designs are likely to be directly applicable to the c2c domain.

6.3.1 Architecture

The architecture of BAD-FS is shown in Figure 6.7. Two types of server processes manage local resources. A *compute server* exports the ability to transfer and execute an ordinary user program on a remote CPU. A *storage server* exports access to disk and memory resources via

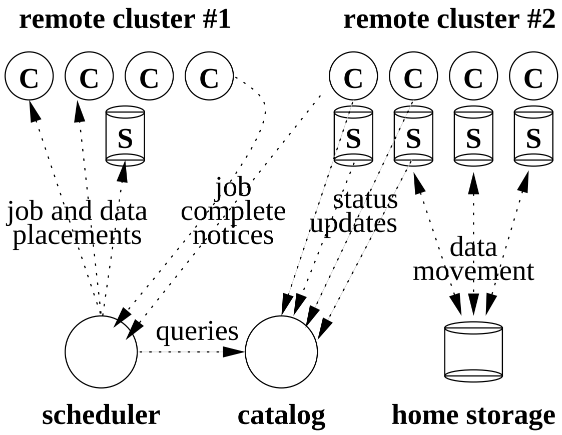


Figure 6.7: Architecture of BAD-FS
Circles are compute servers, which execute batch jobs. Cylinders are storage servers, which hold cached inputs and temporary outputs. Both types of servers report to a catalog server, which records the state of the system. The scheduler uses information from the catalog to direct the system by configuring storage devices and submitting batch jobs.

remote procedure calls that resemble standard file system operations. It also permits remote users to allocate space via an abstraction called *volumes*. An *agent* binds a running process to both compute and storage servers. Both types of servers periodically report themselves to a *catalog*, which summarizes the current state of the system. A *scheduler* periodically examines the state of the catalog, considers the work to be done, and assigns jobs to compute servers and data to storage servers. The scheduler may obtain data, executables, and inputs from any number of external storage sites. For simplicity, I assume the user has all the necessary data stored at a single *home storage server* such as a standard FTP server.

From the perspective of the scheduler, compute and storage servers are logically independent. A specialized device might run only one type of server process: for example, a diskless workstation runs only a compute server, whereas a storage appliance runs only a storage server. However, a typical workstation or cluster node has both computing and disk resources and thus runs both.

BAD-FS may be run in an environment with multiple owners and a high failure rate.

In addition to the usual network and system errors, BAD-FS must be prepared for *eviction* failures in which shared resources may be revoked without warning. The rapid rate of change in such systems creates possibly stale information in the catalog. BAD-FS must also be prepared to discover that the servers it attempts to harness may no longer be available.

BAD-FS makes use of several existing components. The compute servers are Condor [90] *startd* processes, the storage servers are modified NeST storage appliances [26], the interposition agents are Parrot [147] agents, and the catalog is the Condor *matchmaker*. The servers advertise themselves to the catalog via the ClassAd [118] resource description language.

6.3.2 Storage Servers

Storage servers are responsible for exporting the raw storage of the remote sites in a manner that allows efficient management by remote schedulers. A storage server does not have a fixed policy for managing its space. Rather, it makes several policies accessible to external users who may carve up the available space for caching, buffering, or other tasks as they see fit. Using an abstraction called *volumes*, storage servers allow users to allocate space with a name, a lifetime, and a type that specifies the policy by which to internally manage the space. The BAD-FS storage server exports two distinct volume types: scratch volumes and cache volumes.

A *scratch volume* is a self-contained read-write file system, typically used to localize access to temporary data. The scheduler can use scratch volumes for pipeline data passed between jobs and as a buffer for endpoint output. Using scratch volumes, the scheduler minimizes home server traffic by localizing pipeline I/O and only writing endpoint data when a pipeline successfully completes. To permit efficient backup, a storage server can be directed to duplicate a scratch volume onto another server.

A *cache volume* is a read-only view of a home server, created by specifying the name of the home server and path, a caching policy (i.e., LRU or MRU), and a maximum storage size. Multiple cache volumes can be bound into a *cooperative cache volume* by specifying the name of a catalog server, which the storage servers query to discover their peers. Many algorithms [41, 50] exist for managing a cooperative cache, but it is not my intent to explore such algorithms here. Rather, I will give a reasonable algorithm for this system and explain how it is used by the scheduler.

The cooperative cache is built using a distributed hash table [64, 89]. The keys in the table are block addresses, and the values specify which server is primarily responsible for that block. To avoid wide-area traffic, only the primary server will fetch a block from the home server and the other servers will create secondary copies from the primary. When space is needed, secondary data is evicted before primary. To approximate locality, each node only considers hosts on the same IP subnet to be its peers. Thus, each subnet forms a distinct cache.

Failures within the cooperative cache, including partitions, are easily managed but may cause slowdown. Should a cooperative cache be internally partitioned, the primary blocks that were assigned to the now missing peers will be reassigned. As long as the home server is accessible, partitioned cooperative caches will be able to refetch any lost data and continue without any noticeable disturbance to running jobs.

This approach to cooperative caching has two important differences from previous work. First, because data dependencies are completely specified by the scheduler, BAD-FS does not need to implement a cache consistency scheme. Once read, all data are considered current until the scheduler invalidates the volume. This design decision greatly simplifies the implementation; previous work has demonstrated the many difficulties of building a more general cooperative caching scheme [16, 34]. Second, unlike previous cooperative caching schemes that manage cluster memory [41, 50], this cache stores data on local *disks*. Although managing memory caches cooperatively could also be advantageous, the most important optimization to make in our environment is to avoid data movement across the wide-area; managing remote disk caches is the simplest and most effective way to do so.

6.3.3 Interposition Agent

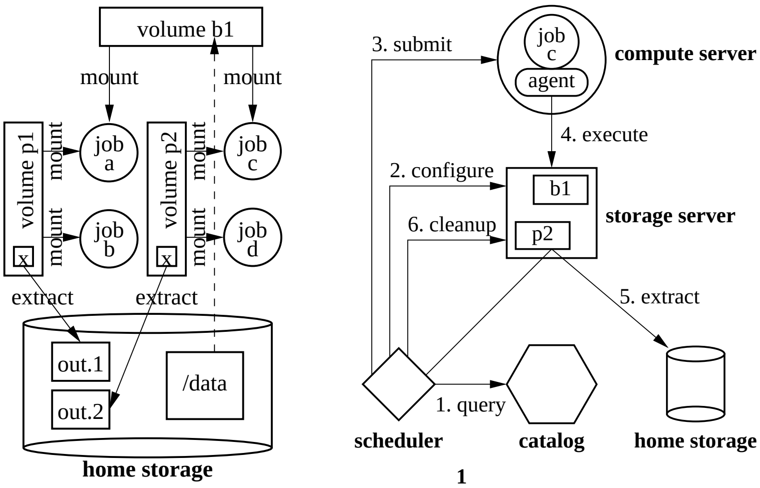
BAD-FS uses Parrot as an agent to connect applications to storage servers. The mapping from logical path names into physical storage is provided by the scheduler at runtime in the form of a mountlist. The notion of error scope introduced in Chapter 5 is critical in this setting because a large number of errors may occur outside of the job's scope. For example, if a volume no longer exists, whether due to accidental failure or deliberate preemption, a storage server returns a unique *volume lost* error to Parrot. Upon discovering such an error, Parrot terminates the job indicating an error of remote resource scope. Explicit error scope allows the scheduler to take transparent recovery actions without returning the job to the user.

6.3.4 The Scheduler

The BAD-FS scheduler directs the execution of a workload on compute and storage servers by combining a static workload description with dynamic knowledge of the system state. Specifically, the scheduler minimizes traffic across the wide-area by differentiating I/O types and treating each appropriately, carefully managing remote storage to avoid thrashing and replicating output data proactively if that data is expensive to regenerate.

Shown in Figure 6.8 is an example of the declarative workflow language that describes a batch-pipelined workload and shows how the scheduler converts this description into an execution plan. The keyword `job` names a job and binds it to a description file, which specifies the information needed to execute that job. The keyword `parent` indicates an ordering between two jobs. The keyword `volume` names the data sources required by the

```
job      a    a.condor
job      b    b.condor
job      c    c.condor
job      d    d.condor
parent   a    child    b
parent   c    child    d
volume   b1   ftp://home/data 1 GB
volume   p1   scratch 50 MB
volume   p2   scratch 50 MB
mount    b1   a    /mydata
mount    b1   c    /mydata
mount    p1   a    /tmp
mount    p1   b    /tmp
mount    p2   c    /tmp
mount    p2   d    /tmp
extract  p1   x    ftp://home/out.1
extract  p2   x    ftp://home/out.2
```



(A) **(B)** **(C)**

Figure 6.8: Workflow and Scheduler Examples.
`(A)` *A simple workflow script. A directed graph of jobs is constructed using* `job` *and* `parent`*, and the file system namespace presented to jobs is configured with* `volume` *and* `mount`*. The* `extract` *keyword indicates which files must be committed to the home storage server after pipeline completion.* `(B)` *A graphical representation of this workflow.* `(C)` *The scheduler's plan for job c. (1) The scheduler queries the catalog for the current system state and decides where to place job c and its data. (2) The scheduler creates volumes b1 and p2 on a storage server. (3) Job c is dispatched to the compute server. (4) Job c executes, accessing its volumes via the agent. (5) After jobs c and d complete, the scheduler extracts x from p2. (6) The scheduler frees volumes b1 and p2.*

workload. For example, volume `b1` comes from an FTP server, while volumes `p1` and `p2` are empty scratch volumes. Volume sizes are provided to allow the scheduler to allocate space appropriately. The `mount` keyword binds a volume into a job's namespace. For example, jobs `a` and `c` access volume `b1` as `/mydata`, while jobs `a` and `b` share volume `p1` via the path `/tmp`. The `extract` command indicates which files of interest must be committed to the home server. In this case, each pipeline produces a file `x` that must be retrieved and uniquely renamed.

Unlike most file systems, BAD-FS is aware of the flow of its data. From the workflow language, the scheduler knows where data originates and where it will be needed. This knowledge allows it to create a customized environment for each job and minimize traffic to the home server. This technique is known as *I/O scoping*.

I/O scoping minimizes traffic in two ways. First, cooperative cache volumes are used to hold read-only batch data such as `b1` in Figure 6.8. Such volumes may be reused without modification by a large number of jobs. Second, scratch volumes, such as `p2` in Figure 6.8, are used to localize pipeline data. As a job executes, it accesses only those volumes that were explicitly created for it; the home server is accessed only once for batch data and not at all for pipeline.

With the workload information expressed in the workflow language, the scheduler neatly addresses the issue of consistency management. All of the required dependencies between jobs and data are specified directly. Since the scheduler only runs jobs so as to meet these constraints, there is no need to implement a cache consistency protocol among the BAD-FS storage servers.

The user may make mistakes in the workflow description that can affect both cache consistency and correct failure recovery. However, through an understanding of the expected workload behavior as specified by the user, the scheduler can easily detect these mistakes and warn the user that the results of the workload may have been compromised. The current implementation does not have these features, but the architecture readily admits them.

Finally, the scheduler makes BAD-FS robust to failures by handling failures of jobs, storage servers, the catalog, and itself. The scheduler keeps a log of allocations in persistent storage, and uses a transactional interface to the compute and storage servers. If the scheduler fails, then allocated volumes and running jobs will continue to operate unaided. If the scheduler recovers, it simply re-reads the log to discover what resources have been allocated and resumes normal operations. Recording allocations persistently allows them to be either re-discovered or released in a timely manner. If the log is irretrievably lost, then the workflow must be resumed from the beginning; previously acquired leases will eventually expire.

In contrast, the catalog server uses soft state. Since the catalog is only used to discover new resources, there is no need to recover old state from a crash. When the catalog is unavailable, the scheduler will continue to operate on known resources, but will not discover new ones. When the catalog server recovers, it rebuilds its knowledge as compute and storage servers send periodic updates.

The scheduler waits for passive indications of failure in compute and storage servers and then conducts active probes to verify. For example, if a job exits abnormally with an error indicating a failure detected by the interposition agent, then the scheduler suspects that the storage servers housing one or more of the volumes assigned to the job are faulty. The scheduler then probes those servers. If all volumes are healthy, it assumes the job encountered transient communication problems and simply reruns it. However, if the volumes have failed or are unreachable for some period of time, they are assumed lost.

The failure of a volume affects the jobs that use it. Running jobs that rely on a failed volume must be stopped. In addition, failures can cascade; completed processes that *wrote* to a volume must be rolled back and re-run in a manner similar to the Time Warp algorithm.

Due to its robust failure semantics, the scheduler need not handle network partitions any differently than other failures. When partitions are formed between the scheduler and compute servers, the scheduler may choose to reschedule any jobs that were running on the other side of the partition. In such a situation, it is possible that the partition could be resolved, at which point the scheduler will find that multiple servers are executing the same jobs. Such overlap will not introduce errors because each job writes to distinct scratch volumes. The scheduler may choose one output to extract and then discard the other.

6.3.5 Practical Issues

One of the primary obstacles to deploying a new distributed system is the need for a compliant administrator. Whether deploying an operating system, a file system, or a batch system, the vast majority of such software requires a privileged user to install and oversee the software. Such requirements make many forms of distributed computing a practical impossibility; the larger and more powerful the facility, the more difficult it is for an ordinary user to obtain administrative privileges. To this end, BAD-FS is packaged as a *virtual batch system* that can be deployed over an existing batch system without special privileges. This technique is similar in structure to the "glide-in job" described by Frey et al. [57] and is similar in spirit to recursive virtual machines [52].

To run BAD-FS, an ordinary user need only to be able to submit jobs into an existing batch system. BAD-FS bootstraps itself on these systems, relying on the basic ability to queue and run a self-extracting executable program containing the storage and compute servers and the interposition agent. Once deployed, the servers report to a catalog server, and the scheduler may then harness their resources. Note that the scheduling of the virtual batch jobs is at the discretion of the host system; these jobs may be interleaved in time and space with jobs submitted by other users. To date, BAD-FS has been deployed over existing Condor and PBS batch systems.

Another practical issue is security. BAD-FS currently uses the Grid Security Infrastructure (GSI) [55], a public key system that delegates authority to remote processes through the use of time-limited proxy certificates. To bootstrap the system, the submitting user must enter a password to unlock the private key at his/her home node and generate a proxy certificate with a user-settable timeout. The proxy certificate is delegated to the remote system and used by the storage servers to authenticate back to the home storage server. Delegation requires that users trust the host system not to steal their secrets, which is reasonable in a cluster-to-cluster environment.

6.4 Performance

Figure 6.9 shows the performance of BAD-FS on the five candidate applications running on a controlled 16-node cluster. The left-hand graphs show an emulated remote cluster where the bandwidth to the home server was constrained at 1 MB/s. The right hand graphs show a local cluster with a home server on the same local area network as other nodes.

Each bar shows the performance of a workload composed of 64 pipelines of the indicated application. Each workload was run in three different configurations:

- **R - Remote I/O.** Each pipeline was equipped by Parrot to access all of its data directly on the remote storage node, with no caching or other use of the storage available on each cluster node.

- **S - Standalone Caches.** Each pipeline was equipped by Parrot to access its data via the local storage node configured as a standard whole-file write-through cache similar to AFS.

- **B - BAD-FS.** All of the BAD-FS machinery.

In the **S** and **B** cases, the workloads ran more efficiently once all of the necessary data were paged in. Thus, the graphs are divided top and bottom to indicate whether caches were cold or warm. The top graphs show the performance of the first 16 jobs running with cold caches, while the bottom graphs show the performance of the remaining 48 jobs running with warm caches. All bars show the parallel efficiency of running each workload in the cluster, compared to running the entire workload sequentially on a single machine equipped with Parrot accessing I/O locally. Thus, a parallel efficiency of 100 percent would indicate a 16x speedup over a sequential run on local data.

From these graphs, several conclusions may be drawn.

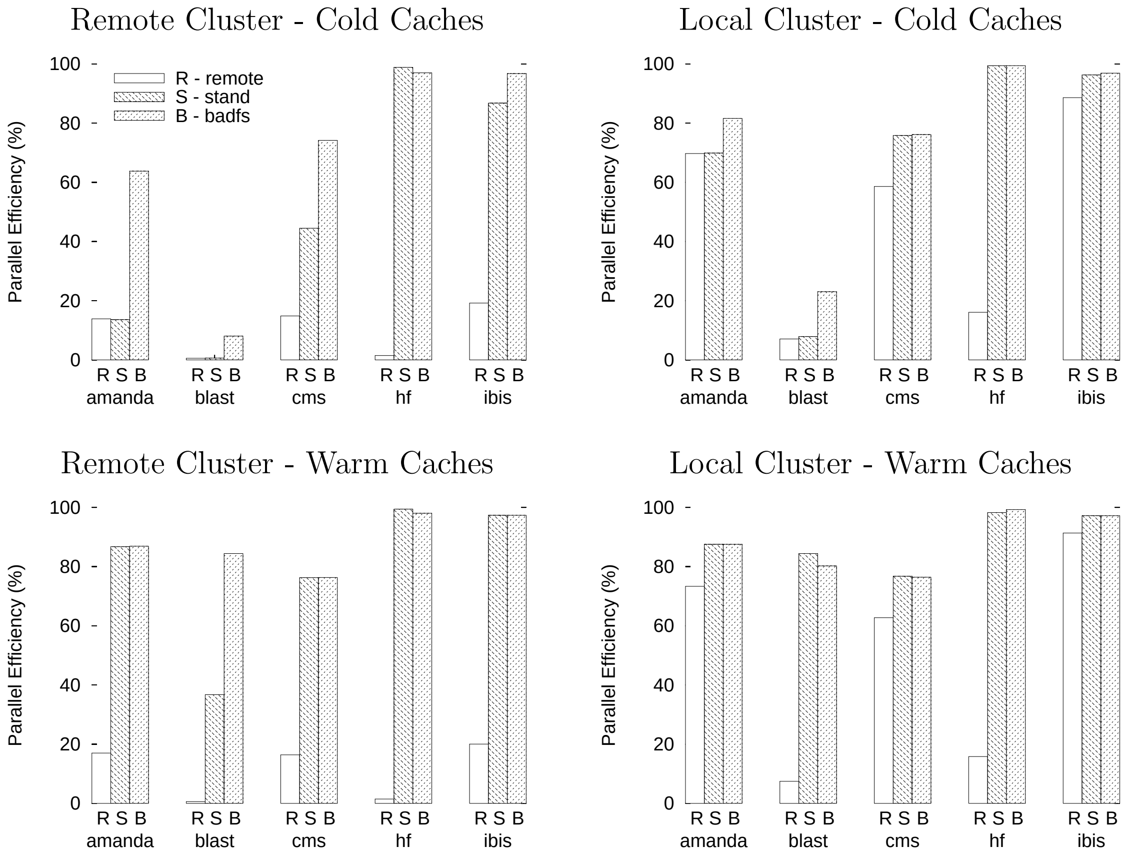


Figure 6.9: Parallel Efficiency of BAD-FS on Real Workloads

First, BAD-FS dramatically exceeds the performance of remote I/O and standalone caching in many cases. In a small number of cases, BAD-FS is slightly slower than standalone caching. These workloads, which are discussed in great detail in the earlier profiling work [143], all have large degrees of either batch or pipeline data sharing. Note that workloads whose I/O consists entirely of endpoint data would gain no benefit from BAD-FS.

Second, the benefit of caching, either cooperatively or in standalone mode, is greater for batch-intensive workloads, such as BLAST, than it is for more pipe-intensive ones such as HF. In these pipe-intensive workloads, the important optimization is I/O scoping, which is performed by both BAD-FS and standalone caching.

Third, cooperative caching in BAD-FS can outperform standalone both during cold and warm phases of execution. If the entire batch data set fits on each storage server, then cooperative caching is only an improvement while the data is being initially paged in. However, should the data exceed the capacity of any of the caches, then cooperative caching, unlike standalone, is able to aggregate the cache space and fit the working set.

This benefit of cooperative caching with warm caches is illustrated in the BLAST measurements in the graph on the left of Figure 6.9. Post-mortem analysis showed that two of the storage servers had slightly less cache space (≈500 MB) than was needed for the total BLAST batch data (≈600 MB). As subsequent jobs accessed these servers, they were forced to refetch data. Refetching it from the wide-area home server in the standalone case was much more expensive than refetching from the cooperative cache as in BAD-FS. With a local-area home server this performance advantage disappears.

Fourth, the penalty for performing remote I/O to the home node is less severe but still significant when the home node is in the same local-area network as the execute cluster. This result illustrates that BAD-FS can improve performance even when the bandwidth to the home server is not obviously a limiting resource.

Finally, comparing across graphs, it can be seen that BAD-FS performance is almost independent of the connection to the home server when caches are cold and becomes independent once they are warm. Using I/O scoping, BAD-FS is able to achieve local performance in remote environments.

6.5 Experience

The real benefit of BAD-FS comes from its ability to operate reliably in the highly dynamic, failure prone environment of the real world. I will illustrate this by detailing a real deployment of BAD-FS.

Two existing batch systems were available for the construction of a combined BAD-FS system. At the University of Wisconsin (UW), a large Condor system of over one thousand CPUs, including workstations, clusters, and classroom machines, is shared among a large number of users. At the University of New Mexico (UNM), a PBS system manages a cluster of over 200 dedicated machines. A personal scheduler, catalog, and home storage server were established at Wisconsin, and then a large number of BAD-FS bootstrap jobs were submitted to both systems without installing any special software at either of the locations. The scheduler was then directed to execute a large workload consisting of 2500 CMS jobs using whatever resources became available. (These CMS jobs were somewhat larger than the benchmark jobs used before. Each job required approximately 30 minutes of CPU time and performed 3.9 GB of batch I/O.)

Figure 6.10 is a timeline of the execution of this workload. As expected, the number of CPUs available to us varied widely, due to competition with other users, the availability of idle workstations (at UW), and the vagaries of each batch scheduler. UNM initially provided twenty CPUs, later jumping to forty after nine hours. Two spikes in the available CPUs between 4 and 6 hours are due to the crash and recovery of the catalog server; monitoring data was lost, but jobs continued to run.

The benefits of cooperative caching are underscored in such a dynamic environment. In the bottom graph, the cumulative read traffic from the home node is shown to have several hills and plateaus. The hills correspond to large spikes in the number of available CPUs.

Whenever CPUs from a new subnet begin executing, they fetch the batch data from the home node. However, smaller hills in the number of available CPUs do not have an effect on the amount of home read traffic because a new server entering an already established cooperative cache is able to fetch most of the batch data from its peers.

Most importantly, this graph shows that harnessing a remote cluster with BAD-FS is effective despite the increased latency and decreased bandwidth associated with the remote cluster, BAD-FS took advantage of the New Mexico cluster for a full four hours before

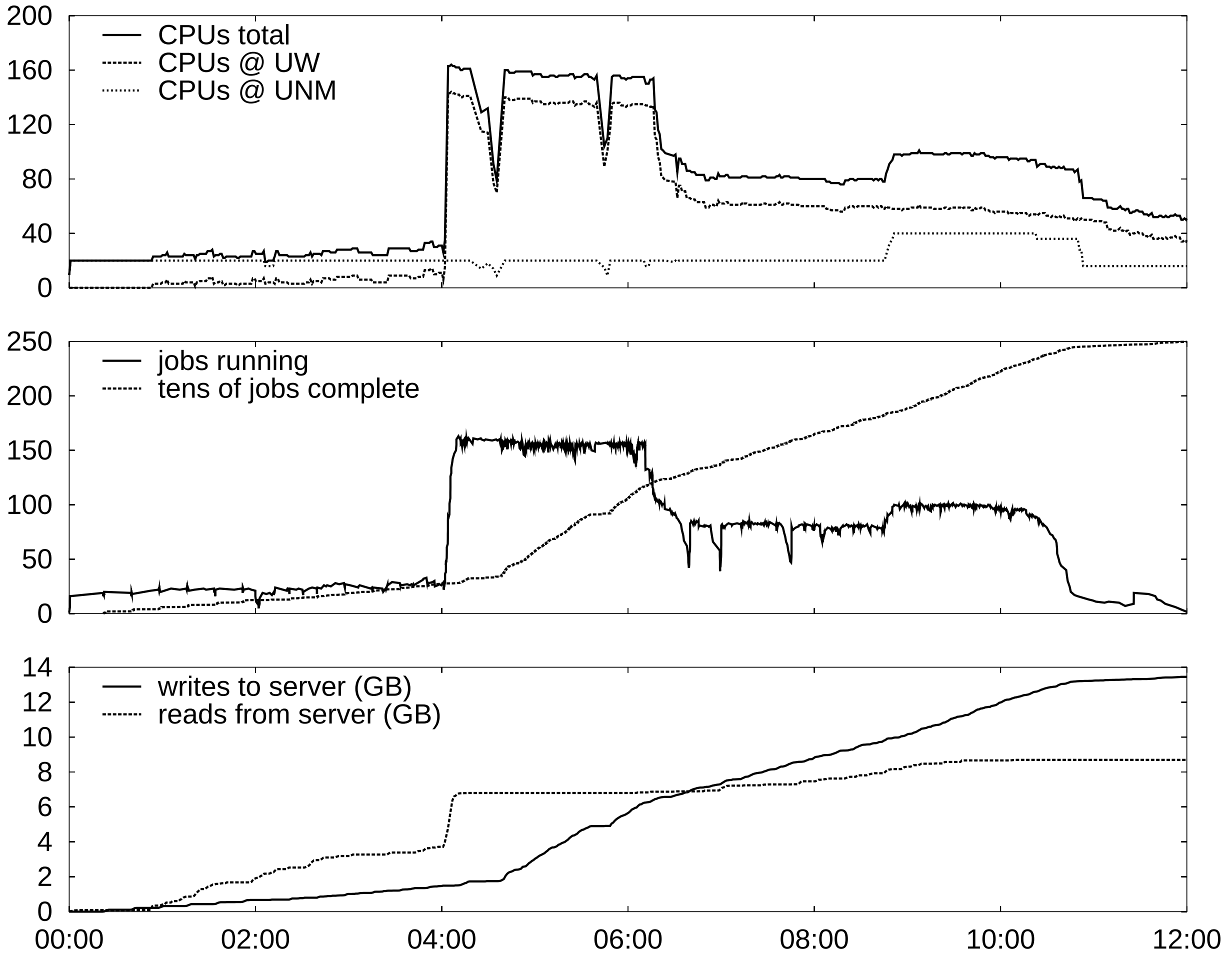


Figure 6.10: Timeline of CMS Workload

Wisconsin was able to provide a significant number of CPUs. In fact, New Mexico completed a significant fraction of the overall workload, despite having fewer, slower CPUs separated from the home storage by a slow network:

	New Mexico	Wisconsin
Jobs Completed	724	1776
Failures	162	828

As these numbers indicate, failures were a significant presence in both clusters. In this context, a failure was counted anytime the scheduler was required to backtrack. Backtracking occurred due to failures to allocate space at storage servers, failures to start or finish a job

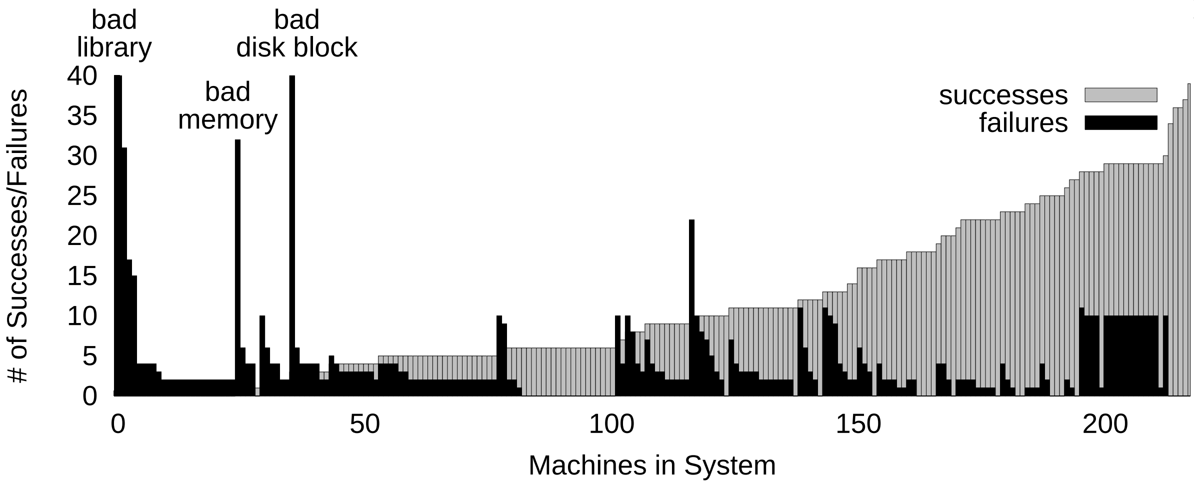


Figure 6.11: Failure Distribution in CMS Workload

at a compute server, and failures to retrieve the outputs of a job from a storage server.

Figure 6.11 details the distribution of failures across the system. Each machine that was involved in the system at some point – 218 total – is enumerated along the horizontal axis. The gray bars indicate the number of jobs successfully completed by that machine. The black bars indicate the number of failures at that machine. For example, machine 218 ran 39 jobs successfully and experienced no failures. Machine 101 ran 6 jobs successfully and experienced 10 failures. The list is sorted by number of successes.

As can be seen, failures are scattered across a wide variety of machines. The productive machines on the right side of the graph generated failures as a side effect of preemption. When the individual batch systems retracted resources from BAD-FS, the scheduler would attempt to re-harness those resources until their information expired from the catalog server. A majority of machines were preempted at some point.

The non-productive machines on the left side of the graph had chronic problems that prevented the completion of any jobs at all. Three machines with a large number of failures are worth mentioning. Machine 0 had an older version of the operating system with a dynamically linked standard library that was slightly incompatible with the CMS application and caused it to crash in the middle of execution. Machine 23 had a faulty memory, while machine 37 had a bad disk block. Both of these caused the failure of a number of storage servers, but also happened to allow a small number of jobs to complete, provided they did not happen to touch the failing address or block in question. These operational problems are good examples of the wide variety of issues that arise daily in operational systems.

One might propose that such failing (i.e. bad memory) and non-ideal (i.e. non-dedicated) machines should be removed from the resource pool so as to "clean up" the system, or at least make it more efficient. This would not be a wise decision, given that BAD-FS derived the *majority* of its computational power from these failing resources. The machines that behaved ideally completed 608 jobs out of the total workload, while the non-ideal machines were responsible for the remaining 1892 jobs.

6.6 Conclusion

BAD-FS brings together all of the concepts and technologies described in this dissertation. In operates on the heterogeneous, unreliable environment and the highly structured applications introduced in Chapter 1. It relies on a sophisticated agent that binds to ordinary Unix programs as in Chapter 3 and mediates the connection to I/O services as in Chapter 4. A crucial component is the ability to understand the scope of an error and communicate that through the batch system as elaborated in Chapter 5. Finally, BAD-FS achieves good performance through I/O scoping, which requires the view of an entire workload as a transaction, described at the beginning of this chapter.

BAD-FS is the union of a batch system and a file system into a coherent whole. By considering computation and data as first-class resources that must be managed together, BAD-FS achieves provides transparent distribution, fault-tolerance, and good performance for data-intensive workloads on complex systems.

Chapter 7

Conclusion

7.1 Recapitulation

The computing world is becoming a complex ecosystem. Despite utopian visions, there is no one system, no one language, no one protocol, no one model of computing that satisfies all users or all system owners. It is simply the reality that computer systems have differentiated themselves into varying roles and languages. Applications must find ways of surviving in this ecosystem, harnessing resources of differing capabilities when and where they become available.

I have advocated agency as the structural solution to this problem. An agent must transform complex, obscure, or specialized interfaces into a form that is usable by a client. An agent must insulate a client from the vagaries of the field by coalescing multiple messages and hiding temporary setbacks. An agent must coordinate multiple external parties so that they come to agreement on a transaction. An agent allows an unmodified program to survive in the harsh world of distributed computing.

I have presented a variety of techniques for coupling a job to an agent. Internal techniques reap a high-performance flexible bond between job and agent, but are ultimately non-portable and undebuggable. External agents create a strict firewall between agent and job, allowing for the clean detection of untrapped operations. Although the per-call overhead of external agency is high, the cumulative effect on real applications is acceptable.

Once the coupling between a job and its agent is established, the agent is then responsible for connecting to the outside world. I have given a concrete discussion of the coupling between an agent and a variety of distributed I/O systems, emphasizing the semantic problems of transforming one interface into another. I have shown that semantic differences create impedance that is manifested in two forms: as a performance penalty and as escaping errors.

An agent must also interact with the services provided by a batch system. Although this interface is much simpler than the I/O interface, the semantics are just as crucial. In particular, the agent is obliged to draw careful distinctions between the classes of errors that it reports. I have introduced the notion of error scope as a method for describing this interface, and presented a discipline for representing and propagating errors.

Finally, the agent must bring computation and data resources together as a coherent whole. The work to be accomplished in a batch system may be thought of as nested transactions. These transactions may be committed at varying granularities, ranging from single I/O operations, all the way up to entire batch workloads. I have presented BAD-FS as a case study of a system that operates at this very large granularity. In particular, BAD-FS permits for the detection and recovery from a wide variety of failures.

7.2 Future Work

7.2.1 Debugging

In this work, I have given the merest suggestion of the universe of failure modes that plague real-world distributed systems. I have described the problem of untrapped operations, of semantic mismatches, of error scopes, and given a quantitative example of the number of failures in a system that ran for 12 hours.

This only scratches the surface. The end users who actually make use of real distributed systems suffer on a daily basis the complexity of using multiple software systems. Version mismatches, installation problems, resource shortages, and software incompatibilities are the normal mode of existence, while a smoothly running production system is a short-lived exception achieved after long hours of what can only be described as *fiddling around.*

The error propagation discipline that I have described allows users of such systems to detect and avoid resources and systems that are failing. Avoidance is sufficient when there exist other resources to be harnessed. Avoidance is an important and valuable step, as the BAD-FS case study shows. However, it does not help the user or administrator that needs to understand why things fail and how they can be repaired.

A valuable debugging tool for such a complex system might operate as follows. System resources could be made to log their state in a structured form. A data collection network could be used to collate the reports of various components over a period of time and collect them in a centralized location. A master debugger could then use this coherent, albeit incomplete, view of the system to produce output like the following:

> You can't execute your job on host X because the credentials delegated from host Y are malformed. This could be because of a time skew between hosts Y and Z (that's happened three times before), or it might be because hosts X and Y have different versions of Kerberos.

Such an output is an ambitious goal. The completeness of such a message is contingent upon the ability to harvest data and may be hampered by privacy concerns and automated reasoning constraints. Regardless, debugging remains one of the single biggest obstacles to the usability of distributed systems.

7.2.2 Delegation

Agency may be viewed from a quasi-legal perspective as a form of delegation. A user or program delegates to an agent the authority to allocate, consume, and dispose of resources on its behalf in order to achieve some goal.

Delegation in computer systems is generally considered as an all-or-nothing affair. Consider a client that performs a traditional remote procedure call to a server. Remote procedure call is a form of delegation in which the client stops what it is doing completely in order to transfer control to the server. Until the server responds with success or failure, the client does nothing. Similarly, in this work, a process delegates all of its computation and data activity to an agent. It trusts it completely.

This complete trust is simplistic. In the real world, we do not trust agents completely: They may have hidden motives, they may execute our wishes incorrectly, or they may make mistakes simply through an error in communication. Exactly the same problems occur in distributed systems. If I delegate a batch job to be executed by a remote batch system, I must consider whether the remote scheduler does not favor my job, whether the remote machines can be trusted to compute the correct result, and whether the job will still exist if a network partition interrupts my supervision.

For many of the reasons described above, a delegation of work to another system may fail. In this sense, delegation is simply a matter of load distribution and not an expression of absolute trust. How are we to build reliable systems and correct computation when we cannot necessarily trust the results computed by participants? How are we to express reservations, requirements, or suspicion regarding system components or sub-programs? These remain open problems.

7.2.3 Programming Models

It is still not quite clear what the most appropriate programming model for distributed science really is. In this work, I have shown that traditional Unix applications can be coupled to distributed systems, modulo some loss in performance and expressiveness due to the impedance matching problems that I have described. But it is not clear that Unix programs are the most appropriate model for distributed computing.

Functional programming has long been advocated [76] as the most natural programming form for distributed computation. This opinion is usually defended by observing that even the most complex collection of scientific applications can be expressed as one monolithic functional program. Any subexpression, so the argument goes, can be split off from the main program and evaluated on a remote node trivially because such a language has no side effects. Failed subexpressions are easily re-computed for the same reasons.

However attractive this model may seem, it has never been seriously applied to large science. This is almost certainly due to the fact that the large majority of programmers, and scientific programmers in particular, are trained in and comfortable with procedural languages. A more fundamental reason is that selecting the appropriate granularity for distribution is a long-standing problem [74, 132, 72] with no fully-general solution. To date, most large-scale computation has relied on manual decomposition of procedural programs.

Explicit message passing environments, such as MPI [153] and PVM [115] have been quite popular in high performance computing environments with reliable resources and fixed parallelism. However, these models break down when resources may be faulty and parallelism is variable. A more fault-tolerant approach to message passing is called *master-worker* or the *bag-of-tasks* approach, where a central work manager doles out parts of a problem to a dynamic set of workers. This master-work approach has been used successfully to hand-code large applications in the SETI [138], Condor Master-Worker [87], and XtremWeb [49] systems. Although this is a natural match for distributed computing, it has also not (yet) proven to be an attractive model for application programmers.

The search for the best language in which to express distributed programs continues. Languages must strike a balance between the comfort of programmers in procedural models and the flexibility of allocation in declarative models. The declarative language of BAD-FS is one example of this balance: programs written in procedural languages are joined together by a declarative language with a fixed distribution granularity. Other combinations are possible, and BAD-FS will certainly not be the last word.

7.3 Postscript

> I would therefore like to posit that computing's central challenge,
> *how not to make a mess of it*, has not yet been met.
> - Edsger Dijkstra

The challenge of computing is the management of complexity. Users wish to run pure applications concerned only with external matters such as atmospheric neutrinos or population growth. Yet, distributed computing threatens this idealism: Software changes, programs crash, and networks fail. An agent sits between these worlds. As an advocate for a user and an application, it carves a safe environment out of the mess of the real world.

Bibliography

[1] The EDA Industry Working Group. http://www.eda.org, 2003.

[2] Globus bug report 950. http://bugzilla.globus.org/globus/show_bug.cgi?id=950, 2004.

[3] The GNU bourne again shell. http://www.gnu.org/software/bash/bash.html, 2004.

[4] The ROOT I/O library. http://root.cern.ch/root, 2004.

[5] Sun Grid Engine. http://wwws.sun.com/software/gridware, 2004.

[6] The Biological Magnetic Resonance Bank. http://www.bmrb.wisc.edu, 2004.

[7] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer Technical Conference*, Atlanta, GA, 1986.

[8] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. FARSITE: Federated, Available, and Reliable Storage for an Incompletely Trusted Environment. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[9] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database Mining: A Performance Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 5(6):914–925, Dec 1993.

[10] Albert Alexandrov, Maximilian Ibel, Klaus Schauser, and Chris Scheiman. UFO: A personal global file system based on user-level extensions to the operating system. *ACM Transactions on Computer Systems*, pages 207–233, August 1998.

[11] William Allcock, Ann Chervenak, Ian Foster, Carl Kesselman, and Steve Tuecke. Protocols and services for distributed data-intensive science. In *Proceedings of Advanced Computing and Analysis Techniques in Physics Research*, pages 161–163, 2000.

[12] J. Almond and D. Snelling. UNICORE: uniform access to supercomputing as an element of electronic commerce. *Future Generation Computer Systems*, 15:539–548, 1999.

[13] S.F. Altschul, W. Gish, W. Miller, E.W. Myers, and D.J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 3(215):403–410, Oct 1990.

[14] D. Anderson, J. Chase, and A. Vahdat. Interposed request routing for scalable network storage. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, 2000.

[15] D. Anderson, K. Yocum, and J. Chase. A case for buffer servers. In *Proceedings of the IEEE Workshop on Hot Topics in Operating Systems (HOTOS)*, April 1999.

[16] T.E. Anderson, M.D. Dahlin, J.M. Neefe, D.A. Patterson, and R.Y. Wang. Serverless Network File Systems. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 109–26, Copper Mountain, CO, Dec 1995.

[17] Ken Arnold and James Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1997.

[18] Algirdas Avizienis and Jean-Claude Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, May 1986.

[19] Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, and John Ousterhout. Measurements of a Distributed File System. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 198–212, Pacific Grove, CA, Oct 1991.

[20] Robert Balzer and Neil Goldman. Mediating connectors. In *19th IEEE International Conference on Distributed Computing Systems*, June 1999.

[21] A. Barak and O. La'adan. The MOSIX multicomputer operating system for high performance cluster computing. *Journal of Future Generation Computer Systems*, 13(4-5):361–372, March 1998.

[22] O. Barring, J. Baud, and J. Durand. CASTOR project status. In *Proceedings of Computing in High Energy Physics*, Padua, Italy, 2000.

[23] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.

[24] Jim Basney, Miron Livny, and Paolo Mazzanti. Utilizing widely distributed computational resources efficiently with execution domains. *Computer Physics Communications*, 140, 2001.

[25] Andy Bavier, Mic Bowman, Brent Chun, David Culler, Scott Karlin, Steve Muir, Larry Peterson, Timothy Roscoe, Tammo Spalink, and Mike Wawrzoniak. Operating system support for planetary-scale services. In *Proceedings of the First Symposium on Network Systems Design and Implementation*, March 2004.

[26] John Bent, Venkateshwaran Venkataramani, Nick LeRoy, Alain Roy, Joseph Stanley, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Flexibility, manageability, and performance in a grid storage appliance. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[27] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gun Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan Eggers. Extensibility, Safety and Performance in the SPIN Operating System. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 267–284, Copper Mountain, CO, Dec 1995.

[28] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *6th Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[29] Andrew D. Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Februrary 1984.

[30] A. P. Black. Exception handling: The case against. Technical Report TR 82-01-02, University of Washington Computer Sciences Department, January 1982.

[31] Gary Deward Brown. *System 390 Job Control Language*. John Wiley and Sons, 1998.

[32] Jason F. Cantin and Mark D. Hill. Cache Performance for Selected SPEC CPU2000 Benchmarks. Computer Architecture News (CAN), Sep 2001.

[33] Vincent Cate. Alex - a global file system. In *The USENIX File System Workshop*, pages 1–12, Ann Arbor, MI, May 1992.

[34] Satish Chandra, Michael Dahlin, Bradley Richards, Randolph Y. Wang, Thomas E. Anderson, and James R. Larus. Experience with a Language for Writing Coherence Protocols. In *Proceedings of the USENIX Conference on Domain-Specific Languages*, Santa Barbara, CA, Oct 1997.

[35] Raymond Chen. Exceptions: Cleaner, more elegant, and wrong. http://weblogs.asp.net/oldnewthing/archive/2004/04/22.aspx, 2004.

[36] David Cheriton. UIO: A uniform I/O system interface for distributed systems. *ACM Transactions on Computer Systems*, 5(1):12–46, February 1987.

[37] Kenneth Chiu, Madhusudhan Govindaraju, and Dennis Gannon. The Proteus multiprotocol library. In *Proceedings of the Conference on Supercomputing*, November 2002.

[38] Phyllis E. Crandall, Ruth A. Aydt, Andrew A. Chien, and Daniel A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, San Diego, CA, 1995.

[39] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, and S. Tuecke. Resource management architecture for metacomputing systems. In *Proceedings of the IPPS/SPDP Workshop on Job Scheduling Strategies for Parallel Processing*, pages 62–82, 1998.

[40] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-Area Cooperative Storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct 2001.

[41] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation (OSDI '94)*, Monterey, CA, Nov 1994.

[42] Jeff Dike. A user-mode port of the Linux kernel. In *Proceedings of the USENIX Annual Linux Showcase and Conference*, Atlanta, GA, October 2000.

[43] C. I. Dimmer. The Tandem non-stop system. In T. Anderson, editor, *Resilient Computing Systems*, chapter 10, pages 178–196. John Wiley and Sons, 1985.

[44] David A. Edwards and Martin S. McKendry. Exploiting Read-Mostly Workloads in The FileNet File System. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles (SOSP '89)*, pages 58–70, Litchfield Park, Arizona, Dec 1989.

[45] K. Ekandham and A. J. Bernstein. Some new transitions in hierarchical level structures. *Operating Systems Review*, 12(4):34–38, 1978.

[46] Margaret Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1992.

[47] Dawson R. Engler, M. Frans Kaashoek, and James W. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 251–266, Copper Mountain, CO, Dec 1995.

[48] Michael Ernst, Patrick Fuhrmann, Martin Gasthuber, Tigran Mkrtchyan, and Charles Waldman. dCache, a distributed storage data caching system. In *Proceedings of Computing in High Energy Physics*, Beijing, China, 2001.

[49] Gilles Fedak, Ccile Germain, Vincent Nri, and Franck Cappello. XtremWeb: A generic global computing system. In *Proceedings of the IEEE Workshop on Cluster Computing and the Grid*, May 2001.

[50] Michael J. Feeley, W. E. Morgan, F. H. Pighin, Anna R. Karlin, and Henry M. Levy. Implementing Global Memory Management in a Workstation Cluster. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 201–212, Copper Mountain, CO, Dec 1995.

[51] J.A. Foley. An integrated biosphere model of land surface processes, terrestrial carbon balance, and vegetation dynamics. *Global Biogeochemical Cycles*, 10(4):603–628, 1996.

[52] Bryan Ford, Mike Hibler, Jay Lepreau, Patrick Tullman, Godmar Back, and Steven Clawson. Microkernels Meet Recursive Virtual Machines. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, Seattle, WA, Oct 1996.

[53] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[54] Ian Foster and Carl Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[55] Ian Foster, Carl Kesselman, Gene Tsudik, and Steven Tuecke. A Security Architecture for Computational Grids. In *Proceedings of the 5th ACM Conference on Computer and Communications Security Conference*, pages 83–92, 1998.

[56] Geoffrey C. Fox and Wojtek Furmanski. Java for parallel computing and as a general language for scientific and engineering simulation and modeling. *Concurrency: Practice and Experience*, 9(6):415–425, 1997.

[57] James Frey, Todd Tannenbaum, Ian Foster, Miron Livny, and Steve Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing*, pages 7–9, San Francisco, California, August 2001.

[58] Tal Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Proceedings of the Network and Distributed Systems Security Symposium*, February 2003.

[59] Douglas P. Ghormley, Devid Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An extensibility system for commodity operating systems. In *Proceedings of the USENIX Technical Conference*, June 1998.

[60] Al Globus, Eric Langhirt, Miron Livny, Ravishankar Ramamurthy, Marvin Solomon, and Steve Traugott. JavaGenes and Condor: Cycle-scavenging genetic algorithms. In *Proceedings of the ACM Conference on Java Grande*, pages 134–139, San Francisco, California, 2000.

[61] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the Sixth USENIX Security Symposium*, San Jose, CA, 1996.

[62] John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12), December 1975.

[63] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, 1993.

[64] Steven D. Gribble, Eric A. Brewer, Joseph M. Hellerstein, and David Culler. Scalable, Distributed Data Structures for Internet Service Construction. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI '00)*, San Diego, CA, Oct 2000.

[65] R. Henderson and D. Tweten. Portable batch system: External reference specification. Technical report, NASA, Ames Research Center, 1996.

[66] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, October 1969.

[67] Koen Holtman. CMS data grid system overview and requirements. CMS Note 2001/037, CERN, Jul 2001.

[68] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.

[69] C. Howell and D. Mularz. Exception handling in large Ada systems. In *Proceedings of the ACM Washington Ada Symposium*, June 1991.

[70] P.O. Hulith. The AMANDA experiment. In *Proceedings of the XVII International Conference on Neutrino Physics and Astrophysics*, Helsinki, Finland, June 1996.

[71] Galen Hunt and Doug Brubacher. Detours: Binary interception of Win32 functions. Technical Report MSR-TR-98-33, Microsoft Research, February 1999.

[72] Galen Hunt and Michael Scott. The Coign Automatic Distributed Partitioning System. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 187–200, New Orleans, Louisiana, Feb 1999.

[73] D. Jefferson, B. Beckman, F. Wieland, L. Blume, M. DiLoreto, P. Hontalas, P. Laroche, K. Sturdevant, J. Tupman, V. Warren, J. Wedel, H. Younger, and S. Bellenot. The time warp operating system. pages 77–93, 1987.

[74] C. J. Jenny. Process partitioning in distributed systems. *IEEE NTC Conference Record*, 2(31):1–10, 1977.

[75] Michael Jones. Interposition agents: Transparently interposing user code at the system interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, pages 80–93, 1993.

[76] S. L. Peyton Jones. Parallel implementations of functional programming languages. *The Computer Journal*, 32:175–186, April 1989.

[77] Burton Kaliski and Yiqun Lisa Yin. On the security of the RC5 encryption algorithm. Technical Report TR-602, RSA Laboratories, September 1998.

[78] J.J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. *Operating Systems Review*, 23(5):213–225, December 1989.

[79] S. Kleiman. Vnodes: An architecture for multiple file system types in Sun Unix. In *Proceedings of the USENIX Technical Conference*, pages 151–163, 1986.

[80] Tevfik Kosar and Miron Livny. Stork: Making data placement a first class citizen in the grid. In *Proceedings of the 24th IEEE Int. Conference on Distributed Computing Systems*, Tokyo, Japan, March 2004.

[81] John Kubiatowicz, David Bindel, Patrick Eaton, Yan Chen, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Westley Weimer, Chris Wells, Hakim Weatherspoon, and Ben Zhao. OceanStore: An Architecture for Global-Scale Persistent Storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, pages 190–201, Cambridge, MA, Nov 2000.

[82] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[83] Tal L. Lancaster. The Renderman Web Site. http://www.renderman.org/, 2002.

[84] S. M. Larson, C. Snow, and V. S. Pande. Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology. In R. Grant, editor, *Modern Methods in Computational Biology*. Horizon Press, 2003.

[85] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. SOCKS protocol version 5. Internet Engineering Task Force (IETF) Request for Comments (RFC) 1928, March 1996.

[86] H. M. Levy. *Capability-Based Computer Systems*. Digital Press, 1984.

[87] Jeff Linderoth, Sanjeev Kulkarni, Jean-Pierre Goux, and Michael Yoder. An enabling framework for master-worker applications on the computational grid. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 43–50, Pittsburgh, Pennsylvania, August 2000.

[88] Barbara Liskov and Alan Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6), 1979.

[89] Witold Litwin, Marie-Anne Neimat, and Donovan Schneider. RP*: A Family of Order Preserving Scalable Distributed Data Structures. In *Proceedings of the 20th International Conference on Very Large Databases (VLDB 20)*, pages 342–353, Santiago, Chile, Sep 1994.

[90] Michael Litzkow, Miron Livny, and Matthew Mutka. Condor - a hunter of idle workstations. In *Proceedings of the Eighth International Conference of Distributed Computing Systems*, June 1988.

[91] Michael Litzkow, Todd Tannenbaum, Jim Basney, and Miron Livny. Checkpoint and migration of Unix processes in the Condor distributed processing system. Technical Report UW-CS-TR-1346, University of Wisconsin - Madison Computer Sciences Department, April 1997.

[92] Michael J. Litzkow. Remote Unix - Turning idle workstations into cycle servers. In *Proceedings of the USENIX Summer Technical Conference*, pages 381–384, 1987.

[93] Lauri Loebel-Carpenter, Lee Lueking, Carmenita Moore, Ruth Pordes, Julie Trumbo, Sinisa Veseli, Igor Terekhov, Matthew Vranicar, Stepher White, and Victoria White. SAM and the particle physics data grid. In *Proceedings of CHEP*, 1999.

[94] Q. Lu and M. Satyanarayanan. Resource conservation in a mobile transaction system. *IEEE Transactions on Computers*, 46(3), March 1997.

[95] Simon Marlow, Simon Peyton Jones, Andrew Moran, and John Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM Conference on Programming Langauge Design and Implementation*, Snowbird, Utah, June 2001.

[96] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the Performance of Log-Structured File Systems with Adaptive Methods. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 238–251, Saint-Malo, France, Oct 1997.

[97] Craig Metz. Protocol independence using the sockets API. In *Procedings of the USENIX Technical Conference*, June 2002.

[98] B. Meyer. *Object-Oriented Software Construction.* Prentice Hall, New Jersey, 1997.

[99] Barton Miller, Mark Callaghan, Jonathan Cargille, Jeffrey Hollingsworth, R. Bruce Irvin, Karen Karavanic, Krishna Kunchithapadam, and Tia Newhall. The Paradyn parallel performance measurement tools. *IEEE Computer*, 28(11):37–46, November 1995.

[100] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing.* MIT Press, Boston, MA, 1985.

[101] Sape Mullender, Guido van Rossum, Andrew Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, 1990.

[102] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen. Ivy: A Read/Write Peer-to-Peer File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[103] R.M. Needham and A.J. Herbert. *The Cambridge Distributed Computing System.* Addison-Wesley, London, 1982.

[104] D. M. Nicol and X. Liu. The dark side of risk (what your mother never told you about time warp). In *Proceedings of the eleventh workshop on Parallel and distributed simulation*, pages 188–195. IEEE Computer Society, 1997.

[105] John Ousterhout, Andrew Cherenson, Frederick Douglis, Michael Nelson, and Brent Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, 1988.

[106] John K. Ousterhout, Herve Da Costa, David Harrison, John A. Kunze, Mike Kupfer, and James G. Thompson. A Trace-Driven Analysis of the UNIX 4.2 BSD File System. In *Proceedings of the 10th ACM Symposium on Operating System Principles (SOSP '85)*, pages 15–24, Orcas Island, WA, Dec 1985.

[107] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *Proceedings of the UKUUG Summer Conference*, London, England, 1990.

[108] Rob Pike, Dave Presotto, Ken Thompson, Howard Trickey, and Phil Winterbottom. The use of name spaces in plan 9. In *Proceedings of the 5th workshop on ACM SIGOPS European workshop*, pages 1–5. ACM Press, 1992.

[109] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.

[110] Platform Computing. Improving Business Capacity with Distributed Computing. www.platform.com/industry/financial/, 2003.

[111] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. LOCUS: A Network Transparent, High Reliability Distributed System. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 169–177, Pacific Grove, CA, Dec 1981.

[112] John Postel. FTP: File transfer protocol specification. Internet Engineering Task Force Request for Comments (RFC) 765, June 1980.

[113] Bruno R. Preiss, Wayne M. Loucks, and Ian D. Macintyre. Effects of the checkpoint interval on time and space in time warp. *ACM Trans. Model. Comput. Simul.*, 4(3):223–253, 1994.

[114] Jim Pruyne. *Resource Management Services for Parallel Applications*. PhD thesis, University of Wisconsin-Madison, 1996.

[115] Jim Pruyne and Miron Livny. Providing resource management services to parallel applications. In *Proceedings of the Second Workshop on Environments and Tools for Parallel Scientific Computing*, May 1994.

[116] Francesco Quaglia. Combining periodic and probabilistic checkpointing in optimistic simulation. In *Proceedings of the thirteenth workshop on Parallel and distributed simulation*, pages 109–116. IEEE Computer Society, 1999.

[117] Mukund Raghavachari and Anne Rogers. Ace: a language for parallel programming with customizable protocols. *ACM Transactions on Computer Systems*, 17(3), August 1999.

[118] Rajesh Raman. *Matchmaking Frameworks for Distributed Resource Management*. PhD thesis, University of Wisconsin, October 2000.

[119] Alastair Reid, Matthew Flatt, Leigh Stoller, Jay Lepreau, and Eric Eide. Knit: Component composition for systems software. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, pages 347–360, San Diego, California, October 2000.

[120] D. Roselli, J.R. Lorch, and T.E. Anderson. A comparison of file system workloads. In *USENIX Annual Technical Conference*, 2000.

[121] Mendel Rosenblum and John Ousterhout. The Design and Implementation of a Log-Structured File System. *ACM Transactions on Computer Systems*, 10(1):26–52, Feb 1992.

[122] Antony Rowstron and Peter Druschel. Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01)*, Banff, Canada, Oct 2001.

[123] John Ruemann and Kang Shin. Stateful distributed interposition. *ACM Transactions on Computer Systems*, 22(1):1–48, February 2000.

[124] Yasushi Saito, Christos Karamanolis, Magnus Karlsson, and Mallik Mahalingam. Taming Aggressive Replication in the Pangaea Wide-area File System. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, MA, Dec 2002.

[125] Asad Samar and Heinz Stockinger. Grid Data Management Pilot. In *In Proceedings of IASTED International Conference on Applied Informatics*, Innsbruck, Austria, February 2001.

[126] Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the USENIX Summer Technical Conference*, pages 119–130, 1985.

[127] M. Satyanarayanan. A Study of File Sizes and Functional Lifetimes. In *Proceedings of the 8th ACM Symposium on Operating Systems Principles (SOSP '81)*, pages 96–108, Pacific Grove, CA, Dec 1981.

[128] Margo Seltzer, Keith Bostic, Marshall Kirk McKusick, and Carl Staelin. An Implementation of a Log-Structured File System for UNIX. In *Proceedings of the USENIX Winter Technical Conference (USENIX Winter '93)*, pages 307–326, San Diego, CA, Jan 1993.

[129] Margo I. Seltzer, Yasuhiro Endo, Christoper Small, and Keith A. Smith. Dealing With Disaster: Surviving Misbehaved Kernel Extensions. In *Proceedings of the 2nd Symposium on Operating Systems Design and Implementation (OSDI '96)*, pages 213–228, Seattle, WA, Oct 1996.

[130] Arie Shoshani, Alex Sim, and Jungmin Gu. Storage resource managers: Middleware components for grid storage. In *Proceedings of the Nineteenth IEEE Symposium on Mass Storage Systems*, 2002.

[131] Abraham Silberschatz and Peter Galvin. *Operating Systems Concepts*. Addison-Wesley, Reading, Massachusetts, 1998.

[132] J.P. Singh and J.L. Hennessy. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *In Proceedings of the International Symposium on Shared Memory Multiprocessing*, Tokyo, Japan, April 1991.

[133] Marvin Solomon and Michael Litzkow. Supporting checkpointing and process migration outside the Unix kernel. In *Proceedings of the USENIX Winter Technical Conference*, pages 283–290, 1992.

[134] Sechang Son and Miron Livny. Recovering internet symmetry in distributed computing. In *Proceedings of the 3rd International Symposium on Cluster Computing and the Grid*, Tokyo, Japan, May 2003.

[135] Mirjana Spasojevic and M. Satyanarayanan. An empirical study of a wide-area distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 14(2):200–222, 1996.

[136] Joel Spolsky. Exceptions. http://www.joelonsoftware.com/items/2003/10/13.html, 2004.

[137] J.G. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Technical Conference*, pages 191–200, 1988.

[138] W. T. Sullivan, D. Werthimer, S. Bowyer, J. Cobb, D. Gedye, , and D. Anderson. A new major SETI project based on project serendip data and 100,000 personal computers. In *Proceedings of the 5th International Conference on Bioastronomy*, 1997.

[139] A. K. Sum and J. J. de Pablo. Nautilus: Molecular Simulations code. Technical report, University of Wisconsin - Madison, Dept. of Chemical Engineering, 2002.

[140] Sam Sun. Establishing persistent identity using the handle system. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001.

[141] Douglas Thain. The Grid Console. http://www.cs.wisc.edu/condor/bypass.

[142] Douglas Thain, Jim Basney, Se-Chang Son, and Miron Livny. The Kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing*, pages 325–333, San Francisco, California, August 2001.

[143] Douglas Thain, John Bent, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, and Miron Livny. Pipeline and batch sharing in grid workloads. In *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*, Seattle, WA, June 2003.

[144] Douglas Thain and Miron Livny. Bypass: A tool for building split execution systems. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, pages 79–85, Pittsburg, PA, August 2000.

[145] Douglas Thain and Miron Livny. Multiple Bypass: Interposition agents for distributed computing. *Journal of Cluster Computing*, 4:39–47, 2001.

[146] Douglas Thain and Miron Livny. Error scope on a computational grid. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, July 2002.

[147] Douglas Thain and Miron Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Proceedings of the Workshop on Adaptive Grid Middleware*, New Orleans, September 2003.

[148] Douglas Thain, Todd Tannenbaum, and Miron Livny. Condor and the grid. In Fran Berman, Geoffrey Fox, and Tony Hay, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[149] Sudharshan Vazhkudai, Steven Tuecke, and Ian Foster. Replica selection in the globus data grid. *IEEE International Symposium on Cluster Computing and the Grid*, May 2001.

[150] Werner Vogels. File System Usage in Windows NT 4.0. In *Proceedings of the 17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 93–109, Kiawah Island Resort, South Carolina, Dec 1999.

[151] D. Waitzman. A standard for the transmission of IP datagrams on avian carriers. Internet Engineering Task Force (IETF) Request For Comments (RFC) 1149, April 1990.

[152] B. White, A. Grimshaw, and A Nguyen-Tuong. Grid-Based File Access: The Legion I/O Model. In *Proceedings of the Ninth IEEE Symposium on High Performance Distributed Computing*, August 2000.

[153] Derek Wright. Cheap cycles from the desktop to the dedicated cluster: combining opportunistic and dedicated scheduling with Condor. In *Conference on Linux Clusters: The HPC Revolution*, Champaign-Urbana, Illinois, June 2001.

[154] E. Zadok and J. Nieh. FiST: A language for stackable file systems. In *The USENIX Annual Technical Conference*, San Diego, California, June 2000.

[155] Victor Zandy and Barton Miller. Reliable network connections. In *Proceedings of the Eighth ACM International Conference on Mobile Computing and Networking*, pages 95–106, Atlanta, GA, September 2002.

[156] Victor Zandy, Barton Miller, and Miron Livny. Process hijacking. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.

[157] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a load sharing facility for large, heterogenous distributed computer systems. *Software: Practice and Experience*, 23(12):1305–1336, December 1993.