Error Scope on a Computational Grid: Theory and Practice

Douglas Thain and Miron Livny
University of Wisconsin, Computer Sciences Department
E-mail: thain,miron@cs.wisc.edu

Abstract

Error propagation is a central problem in grid computing. We re-learned this while adding a Java feature to the Condor computational grid. Our initial experience with the system was negative, due to the large number of new ways in which the system could fail. To reason about this problem, we developed a theory of error propagation. Central to our theory is the concept of an error's scope, defined as the portion of a system that it invalidates. With this theory in hand, we recognized that the expanded system did not properly consider the scope of errors it discovered. We modified the system according to our theory, and succeeded in making it a more robust platform for distributed computing.[1]

1. Introduction

The Condor distributed batch system [29, 42, 17] is software for managing a computational grid [30, 14, 21]. Recently, we extended Condor with a feature that supports the execution of Java [4] programs coupled with secure access to remote storage. The result is a worldwide distributed system providing a uniform environment for both execution and I/O.

Contrary to some expectations, this change entailed much more than simply placing the keyword `java` in front of the program name. The introduction of the Java Virtual Machine (JVM) and Condor I/O facilities created a large new set of failure modes. We found it difficult to manage this new large set of errors, especially as they passed through the many software layers in the system. More importantly, our users were frustrated to be exposed to many of these errors.

In order to reason about this problem, we turned to the standard literature on fault tolerance [1, 39, 18]. We found plenty of advice on how to replicate a computation so as to reduce the probability of failure, but we found little guidance on how to represent and propagate errors through autonomous layers of software. This surprised us, given that layering is a well-established concept [11, 45], that errors have been identified as an obstacle to user interaction [12, 24], and that many languages provide specialized error constructs such as exceptions [32, 23, 27].

Naturally, error propagation is well-understood when all the components of a system are available for inspection and modification. If the addition of a new component in a lower layer introduces a new error type, we simply work our way up through each layer, adding cases for the new error as necessary. However, in a computational grid composed of multiple autonomous components, we do not have the luxury of re-engineering every component when a new one is introduced. How should we deal with new error modes in such a system?

To address this problem, we have constructed a theory of error propagation suitable for a computational grid. We will begin by describing our architecture for executing Java programs and discuss the difficulties we encountered. We then develop our theory of error propagation by building on concepts from fault-tolerance and software engineering. This analysis yields several succinct guiding principles. Finally, we return to the problem presented by Condor and apply our principles. We conclude with some reflections on the applicability of this theory.

2. Architecture

To discuss the problem of error propagation in a complex system, we are obliged to detour long enough to describe our system in moderate detail. The reader familiar with the Condor terminology may comfortably skim ahead, while the reader interested in more detail may consult the references.

[1]This research was supported in part by a Cisco Distinguished Graduate Fellowship.

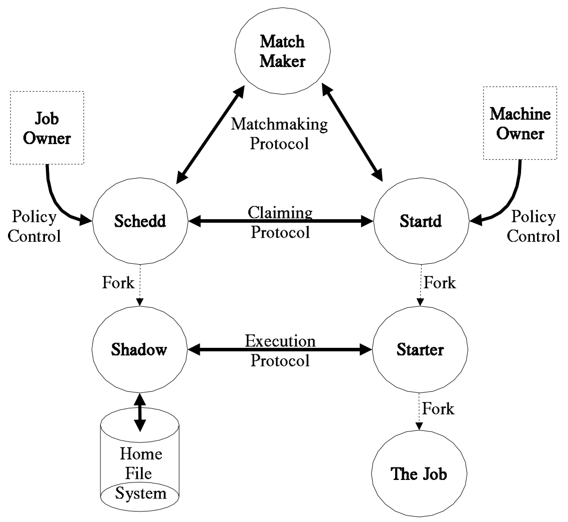

Figure 1. The Condor Kernel

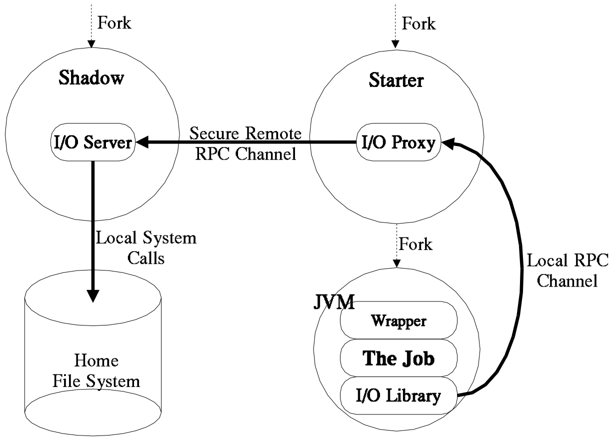

Figure 2. The Java Universe

2.1. Condor Overview

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. 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 [28], and so is uniquely prepared to deal with an unfriendly execution environment by using tools such as process migration [42] and transparent remote I/O [29].

The core components of Condor, known as the Condor Kernel, are shown in Figure 1. They works as follows: 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 unique language known as ClassAds [38], 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 a scratch directory, loading the executable, and moving input and output files.

Condor provides several *universes* for executing jobs. Each universe provides a package of environmental features. 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 outside of a shared file system. Specialized universes are provided for the Globus [17], PVM [37], and MPI [48] environments.

2.2. The Java Universe

A growing number of computational scientists are turning to Java as a suitable language for distributed computing [16, 19]. Although programs written for the JVM 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, we have added a Java Universe to Condor. The added components are shown in Figure 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 transfers the user'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, we provide a simple I/O library. This library presents files using standard Java abstractions, such as the `InputStream` and `OutputStream` interfaces.

This library does not communicate directly with any storage resource, but instead calls a proxy in the starter via a simple protocol called Chirp. 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 I/O functionality to the job without placing any burden on the user. We envision that security and discovery will be the typical applications of this proxy. For example, a local firewall or other security device may be crossed using information known only to the proxy, such as port numbers and security credentials. Or, the proxy may select an appropriate I/O device using a replica management system [46] and transfer data using a secure, high-performance mechanism such as GridFTP [3].

We demonstrate 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 [15] or Kerberos [43].

2.3. Initial Experience

Our initial experience with this design was quite disappointing. 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.

We found this frustrating, as we had gone to great trouble to assure that no error value was left unconsidered. For example, we 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.

3. A Theory of Error Propagation

To better understand this problem, we require a theory of error propagation. We will first describe key concepts relating to errors and then embark on a discussion yielding several succinct design principles. Our goal is not to design new algorithms for fault-tolerant systems. Rather, we 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.

3.1. Terms

The generally accepted definitions of fault, error, and failure are those given by Avizienis and Laprie [5]. 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 random cosmic ray that passes through the machine and corrupts some storage would be a fault. 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.

An error may be communicated in one of three ways:

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 of 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 space." 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.

Both explicit and escaping errors have been represented in recent languages by the *exception* [20]. 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. This permits the implementation of both explicit and escaping errors.

The exception is a useful programming tool, and we are 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. We 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.

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 return value 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 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 raise the specter of 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 contractual 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.

The escaping error represents the mismatch between an interface and an implementation. A 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 [47].

3.3. Error Scope

To be accepted by end users, grid software must be sensitive to the distinction between the explicit and the escaping error. If the grid 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 grid 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 a failure in terms inexpressible in a given interface. To solve this problem, we need an abstraction that balances the diagnostic need for information with the principle of separation between implementation and interface.

We introduce the abstraction of error scope to solve this problem. The *scope* of an error is 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) [6] has process scope. It indicates that the mechanism of function call is no longer valid within the process. A node failure in PVM [37] 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 in Section 4.

3.4. Generic Errors

A frequent source of confusion in error propagation is the generic error. A *generic error* is an indication that an 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 innocuous 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 encourage 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 standard 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. Will it throw a `DiskFull` or a `FullDisk`? 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`.

We 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. We 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 we were able to revise these I/O interfaces to conform to Principle 4, we 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.

4. Condor Revisited

To apply these ideas to the Java Universe, we must first identify the system's various error scopes and their handling programs. This is shown in Figure 3. Dotted lines indicate scopes, circles indicate handling programs, squares indicate resources, and arrows indicate return values.

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 can 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, our mistake in building the Java Universe becomes clear: We 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. We will give two examples.

While executing a Java program, we relied entirely on the exit code of the JVM as an indicator of program success. This introduced implicit errors, violating Principle 1. As Figure 4 shows, the JVM can cause an environmental error to appear as a program result. To retrieve the necessary information, we added the *wrapper* shown in Figure 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, we blindly converted all possible explicit errors from the proxy directly into corresponding Java exceptions. For failure modes not represented by existing exception types, we simply extended the basic `IOException` to a new type. Although this was easy, it was incorrect. We 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, we applied Principle 2 and modified the I/O library to send an escaping error (a Java `Error`) to the program wrapper, which communicates the

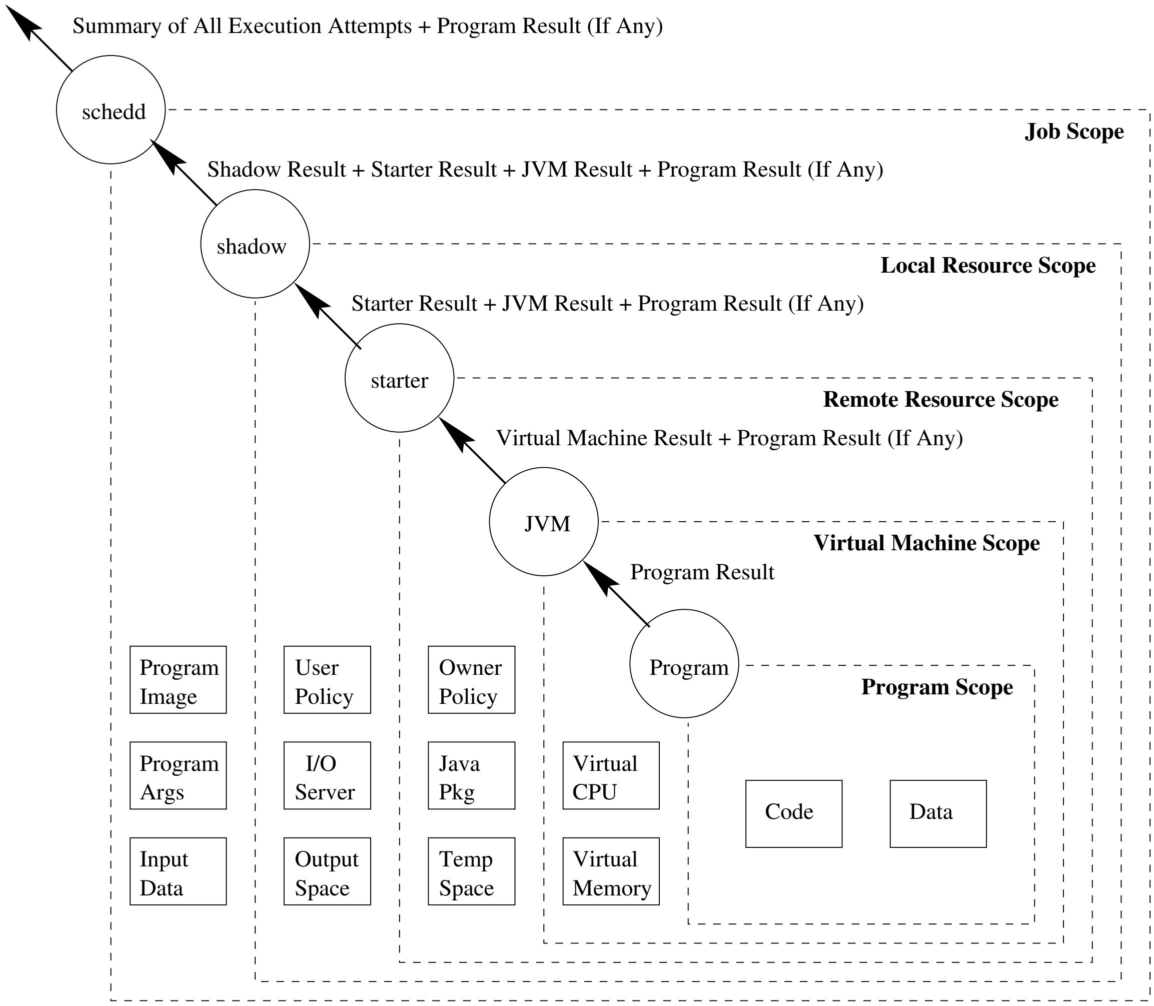

Figure 3. Error Scopes in the Java Universe

This figure shows the various error scopes and their 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. Labelled arrows indicate return values from one handler to another.

Execution Detail	Error Scope	JVM Result Code
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 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 in Section 4.

error scope to the starter through the result file.

The reader may object that the “reasonable expectations” test is more suited to the courtroom than the machine room. We admit that there is room for disagreement in such a subjective term. It is precisely this confusion which motivates our statement of Principle 4.

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

5. Other Directions

Our theory of error propagation brings some structure to the analysis of errors in complex systems, particularly as information crosses system boundaries. Of course, there remain many open issues in the problem of error propagation.

A high frequency of errors delivered correctly may still be a performance problem. For example, a small number of misconfigured machines in our Condor pool attracted a continuous stream of jobs that would attempt to execute, fail, and be returned to the schedd. Although the situation was handled correctly, there was continuous waste of CPU and network capacity. To rectify this, we borrowed a lesson from the Autoconf [31] tool. Rather than blindly accept each owner’s assertion regarding the Java installation, we modified the startd to test the installation at startup. If found lacking, then the startd simply declines to advertise its Java capability. A complementary approach would be to enhance the schedd with logic to detect and avoid hosts with chronic failures.

The appropriate response to an error may be unclear if its scope is indeterminate. This problem is particularly common in networking. For example, a refused network connection may indicate that the target service is temporarily offline, or it may indicate that the caller has given an invalid address. In these situations, time becomes a factor in error propagation. A failure to communicate for one second may be of network scope, but a failure to communicate for a year likely has larger scope. To distinguish between the two, a system must be given some guidance in the form of timeouts or other resource constraints from the user or administrator. A example of this problem is found in NFS [41], where a file system may either be “hard mounted” to hide all network errors or “soft mounted” to expose them to callers after a certain retry period expires. Both users and administrators routinely comment how both of these choices are unsavory, as they offer no mechanism for a single program to choose its own failure criteria.

We have concentrated on explicit errors, but implicit errors are a more difficult topic, and may be more common than we care to believe. Despite low-level error correction, implicit errors have been observed in increasingly uncomfortable rates in networks, memories, and CPUs [44, 34]. Condor itself has little recourse for discovering such errors in applications unless it knows *a priori* the structure of a job or its valid inputs and outputs. The end-to-end principle [40] tells us that the ultimate responsibility for detecting such errors lies with a higher level of software. A process above Condor may work on behalf of the user to analyze outputs and replicate or resubmit jobs that fail due to implicit errors or failures in Condor itself.

6. Related Work

The Reflective Graph and Event (RGE) model [36, 35], developed by the Legion [21] project, is a significant effort to address fault-tolerance in widely distributed systems. In this model, a standard library is used to implement all data structures and communication in all components of the system. Fault-tolerance is then achieved through modeling and introspection. A program may detect faults by comparing reality to a model of its expected behavior. Several forms of fault-tolerance may be implemented by inspecting and modifying one’s own data structures during execution. This approach yields great power within a single closed system where all components are available for introspection. In Condor, if the starter were aware of all the internal details of a remote JVM, then the significance of an execution error would be obvious by design. However, reality dictates that Condor and Java hide implementation details from each other. Our approach complements the RGE model by considering how errors propagate between closed systems.

The use of exceptions as a language feature is generally attributed to Goodenough [20], and has progressed through a variety of languages from research to commercial use, including CLU [27], Haskell [32] Ada [23], C++ [13], and Java [4], to name a few. However, the exception has not received universal approbation; see Black [7] for an argument against. The many variations on the exception concept have disagreed on whether an interface must declare all possible exceptions present in the implementation.

The discipline of *design by contract*, proposed abstractly by Hoare [22] and developed concretely by Meyer[33], offers some discussion of the distinction between explicit and escaping errors. Similar hints are also given by Goodenough [20], Ekanadham and Bernsteien [26], and Howell and Mularz [23]. In such work, the escaping error is usually implemented by an instruction that brings the entire computation to a halt with a message to the console. This is neither possible nor desirable in a distributed system. We build upon this work by arguing that escaping errors must be studied, expected, and structured.

A universal instruction set for heterogeneous computing has been a persistent goal of computer science for many decades. Diehl et al. [10] offer a bibliography of such systems. The Java Virtual Machine [4] has recently been the favorite target for a variety of distributed computing systems [9, 8, 2]. An early exploration of Java support for Condor [19] examined primarily the problem of transparent checkpointing. One major obstacle to the acceptance of Java in scientific computing is a concern about the precise semantics of floating-point operations [25].

7. Conclusion

We have used the Condor Java Universe as a detailed example for exploring a theory of error propagation. The main contributions of our theory are several succinct design principles and the concept of error scope. In summary, our principles are:

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

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

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

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

Our initial design mistake in the Java Universe was to aggressively represent all possible faults as explicit errors in the nearest interface. Our redesign recognized that errors violating the assumptions of the caller must instead be consumed by the surrounding system. The necessary changes were small but powerful.

The scope of an error is an abstraction that allows cooperating processes to take appropriate action without understanding the full detail of an error. In Condor, the submission site does not understand all of the possible reasons a program may fail to execute, nor does the execution site understand all of the possible reasons an I/O operation may fail. Yet, the two may cooperate by knowing the scope, rather than the detail, of errors that they communicate.

In conclusion, we mentioned that our first implementation was correct in the sense that users received true information. If the end-to-end principle obliges the user to analyze a program's outputs anyway, then why must we go to all this trouble to analyze errors? The answer is, of course, performance of the most coarse variety. A human is the slowest part of any computing system. A disciplined error propagation system conserves two precious resources: time and aggravation.

8. Acknowledgments

The Condor software is the work of many people. Although we may claim credit (or blame!) for the most recent changes, much of the understanding of failure modes comes from the first-hand experience of its many contributors. We would like to acknowledge the dozens of people who have added code and understanding, and especially recognize the core architects Michael Litzkow, Todd Tannenbaum, and Derek Wright. We would also like to thank Remzi Arpaci-Dusseau, Alain Roy, Brian White, and the anonymous referees for their helpful advice while preparing this paper.

References

[1] R. J. Abbot. Resourceful systems for fault tolerance, reliability, and safety. *ACM Computing Surveys*, 22(1), March 1990.

[2] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. SuperWeb: research issues in Java-based global computing. *Concurrency: Practice and Experience*, 9(6):535–553, 1997.

[3] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke. Protocols and services for distributed data-intensive science. In *Proceedings of Advanced Computing and Analysis Techniques in Physics Research (ACAT)*, pages 161–163, 2000.

[4] K. Arnold and J. Gosling. *The Java Programming Language, Second Edition*. Addison-Wesley, Reading, Massachusetts, 1997.

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

[6] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, Februrary 1984.

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

[8] H. Casanova and J. Dongarra. Netsolve: A network solver for solving computational science problems. Technical Report CS-95-313, University of Tennessee, Department of Computer Science, 1995.

[9] K. M. Chandy, B. Dimitrov, H. Le, J. Mandelson, M. Richardson, A. Rifkin, P. A. G. Sivilotti, W. Tanaka, and L. Weisman. A world-wide distributed system using Java and the Internet. In *Proceedings of the Fifth IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Syracuse, New York, August 1996.

[10] S. Diehl, P. Hartel, and P. Sestoft. Abstract machines for programming language implementation. *Future Generation Computer Systems*, 16:739–751, 2000.

[11] E. W. Dijkstra. The structure of the THE multiprogramming system. In *Proceedings of the ACM Symposium on Operating System Principles (SOSP)*, Gatlinburg, Tennessee, October 1967.

[12] K. Efe. A proposed solution to the problem of levels in error-message generation. *ACM Computing Practices*, 30(11):948–955, November 1987.

[13] M. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*. Addison Wesley, 1992.

[14] I. Foster and C. Kesselman. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[15] 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.

[16] G. C. Fox and W. 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.

[17] J. Frey, T. Tannenbaum, I. Foster, M. Livny, and S. Tuecke. Condor-G: A computation management agent for multi-institutional grids. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC)*, pages 7–9, Sffan Francisco, California, August 2001.

[18] F. C. Gartner. Fundamentals of fault-tolerance distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1), March 1999.

[19] A. Globus, E. Langhirt, M. Livny, R. Ramamurthy, M. Solomon, and S. Traugott. JavaGenes and Condor: Cycle-scavenging genetic algorithms. In *Proceedings of the ACM Conference on Java Grande*, pages 134–139, San Francisco, California, 2000.

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

[21] A. Grimshaw, W. Wulf, et al. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

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

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

[24] E. J. S. Jr. Making APL error messages kinder and gentler. In *ACM Conference on APL*, pages 320–324, 1989.

[25] W. Kahan and J. Darcy. How Java's floating-point hurts everyone everywhere. Talk given at the ACM 1998 Workshop on Java for High-Performance Network Computing (http://www.cs.uscb.edu/conferences/˜wkahan/JAVAhurt.pdf), March 1998.

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

[27] B. Liskov and A. Snyder. Exception handling in CLU. *IEEE Transactions on Software Engineering*, 5(6), 1979.

[28] M. Litzkow, M. Livny, and M. Mutka. Condor - a hunter of idle workstations. In *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[29] M. J. Litzkow. Remote UNIX - Turning idle workstations into cycle servers. In *USENIX Conference Proceedings*, pages 381–384, Summer 1987.

[30] M. Livny and R. Raman. High-throughput resource management. In I. Foster and C. Kesselman, editors, *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[31] D. Mackenzie, R. McGrath, and N. Friedman. Autoconf: Generating automatic configuration scripts. http://www.gnu.org/autoconf, 1994.

[32] S. Marlow, S. P. Jones, A. Moran, and J. Reppy. Asynchronous exceptions in Haskell. In *Proceedings of the ACM Conference on Programming Langauge Design and Implementation (PLDI)*, Snowbird, Utah, June 2001.

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

[34] D. Milojicic, A. Messer, J. Shau, G. Fu, and A. Munoz. Increasing relevance of memory hardware errors: A case for recoverable programming models. In *Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.

[35] A. Nguyen-Tuong. *Integrating Fault-Tolerance Techniques in Grid Applications*. PhD thesis, University of Virginia, August 2002.

[36] A. Nguyen-Tuong and A. S. Grimshaw. Using reflection for incorporating fault-tolerance techniques into distributed applications. *Parallel Processing Letters*, 9(2):291–301, 1999.

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

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

[39] B. Randell, P. Lee, and P. Treleaven. Reliability issues in computing system design. *ACM Computing Surveys*, 10(2), June 1978.

[40] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[41] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network filesystem. In *Proceedings of the Summer 1985 USENIX Conference*, pages 119–130, 1985.

[42] M. Solomon and M. Litzkow. Supporting checkpointing and process migration outside the UNIX kernel. In *USENIX Conference Proceedings*, pages 283–290, Winter 1992.

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

[44] J. Stone and C. Partridge. When the CRC and TCP checksum disagree. In *Proceedings of ACM SIGCOMM*, Stockholm, Sweden, August 2000.

[45] A. Tannenbaum. *Structured Computer Organization*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[46] S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. *IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2001.

[47] 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.

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