The Ethernet Approach to Grid Computing

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

Abstract

Despite many competitors, Ethernet became the dominant protocol for local area networking due to its simplicity, robustness, and efficiency in wide variety of conditions and technologies. Reflecting on the current frailty of much software, grid and otherwise, we propose that the Ethernet approach to resource sharing is an effective and reliable technique for combining coarse-grained software when failures are common and poorly detailed. This approach involves placing several simple but important responsibilities on client software to acquire shared resources conservatively, to back off during periods of failure, and to inform competing clients when resources are in contention. We present a simple scripting language that simplifies and encourages the Ethernet approach, and demonstrate its use in several grid computing scenarios, including job submission, disk allocation, and data replication. We conclude with a discussion of the limitations of this approach, and describe how it is uniquely suited to high-level programming.

1. Introduction

Any user or designer of distributed systems knows from experience that failures are endemic to large systems. They occur at every level from individual transistors all the way up to high-level software. Ordinary workstations fail or reboot about once a day. [18] One in several thousand TCP segments fails its checksum. [25] One estimate suggests large commodity clusters experience several memory errors a day despite error-correcting hardware. [21]

This is doubly true for grid computing. Those involved in the day-to-day activities of deploying and operating such systems need not be reminded of the tyranny of failures, frequently from sources that are entirely unexpected. Although well-known software tools and techniques carefully manage capital investments such as data archives and CPU clusters, the source of failures is frequently in some prosaic unmanaged resource such as free file descriptors or free scratch space in a user's home directory.

In particular, at high levels of abstraction in integration tools such as command line shells, the necessary details to analyze and react precisely to such errors simply are not available. This is not the fault of any single component, but contributors include blunt software interfaces, conventions designed for interactive users, and sometimes the essential nature of distributed systems. Rather than suggest that we throw out such software and start over, we want to investigate how such tools may be made expressive, efficient, and reliable.

To attack this problem, we re-use a good idea: the Ethernet arbitration protocol. The Ethernet protocol is not simply endless retries, but is a set of precise obligations placed upon clients that encourage the efficient use of a shared resource without worrying too much about precisely what the source of a problem may be.

Further, we propose that the Ethernet idea should not simply be a hidden implementation technique, but should be exposed to both users and administrators at the highest layers of programming. To explore this idea, we present a simple language, the **fault tolerant shell**, which exposes untyped failures in a manner similar to exceptions in other languages. For example, this fragment retries a program for up to one hour in three different configurations for five minutes each:

```
try for 1 hour
    forany host in xxx yyy zzz
        try for 5 minutes
            fetch-file $host filename
        end
    end
end
```

This language is easily implemented and greatly relieves the user of dealing with the complexity of many error conditions. However, we must caution that a cavalier attitude toward the *reason* for an error must not extend to the details necessary for recovery. As we will show, there are subtle details to the Ethernet approach that must be obeyed in order to achieve efficient allocation of shared resources, especially when contention is unexpected.

With this technique, we will demonstrate three grid computing scenarios where conventional tools fail, but the Ethernet approach succeeds. These include submitting jobs to a scheduler, sharing a filesystem as an output buffer, and reading data from potentially faulty servers.

We must state up front that the Ethernet approach is not applicable in all cases or at all levels of a system. It is most useful in uncontrolled systems with highly variable performance, a description that applies to many wide-area computing systems. It is less applicable in tightly-controlled, centralized systems. After describing the details of the Ethernet approach, we will conclude with some discussion of its applicability.

2. Programming the Grid

> *Pascal is for building pyramids – imposing, breathtaking, static structures built by armies pushing heavy blocks into place. Lisp is for building organisms. - Alan J. Perlis [1]*

Today, deploying a grid is like building pyramids with stone and mortar. Most applications destined for grid computing are not specialized for or even aware of their role. Rather, large existing codes, written in languages such as C, Fortran, or Java, are adapted into a grid environment through glue languages such as Perl, Tcl, or Bourne shell. Glue programs serve to insulate applications from the system, preserving the illusion of a reliable standalone system.

These glue languages are designed for rapid prototyping, system assembly, and user interface, and they are quite good at those tasks. However, they are not as well suited for grid computing. Grid jobs are generally not interactive. They may operate for hours, days, or months without direct connection to a living user. Further, they are likely to operate in environments that are wildly different than the original designer used. Glue programs used in this environment appear plausible at first, but then fall to pieces in a grid environment. There are (at least) two major causes of this problem: timing is uncontrollable, and failures lack detail. Let's consider each in detail.

Timing is Uncontrollable. The timing of arbitrary programs is far beyond the control of the ordinary user of a modern computing system. Consider a NFS [23] distributed file system. The client interface of most NFS implementations does not include any way for the user to specify how long an operation may be retried before it is considered a failure. This parameter is left to the system administrator and typically has two values. A "soft-mounted" file system indicates failure to the application after retries exceed sixty seconds. A "hard-mounted" file system retries forever, never admitting failure to the application.

These two settings are unlikely to satisfy all users. Some users doing high-throughput batch processing may be perfectly happy to suffer a delay of up to a day simply for the convenience of unsupervised recovery. Others performing interactive work may wish to be exposed to failures after five seconds so that work may be retried elsewhere.

Such delays are not unique to NFS, or even simply to distributed file systems. Other delays may be forced while waiting for shared locks, for software licenses, for available disk space, or for network bandwidth. Indeed, unexpected delays are the norm for any computing resource that is chosen from a heterogeneous group, is shared among multiple users, or is simply unreliable. A fundamental operation may take a second, or it may take an hour: the user does not know.

Failures Lack Detail. Systems built out of POSIX processes exchange very little information at the interface between programs. POSIX programs are permitted to exit in one of two ways: normally and abnormally. Normal termination leaves behind an integer result whose meaning is program-specific, while abnormal termination leaves behind a signal number indicating a system-level problem such as "segmentation fault" or "division by zero." This division of error types is somewhat analogous to a function in an exception-oriented language that may return an ordinary integer or one of a number of exceptions. The former indicates that the function successfully computed a result, while the latter indicates no result could be computed.

We might be tempted to use this distinction to carry something like typed exceptions between processes. However, the reality of POSIX programs makes this impractical. Consider the command line `cp a b`, which invokes the command `cp` to copy the file named `a` to the file named `b`. There are many ways for this command to succeed or fail, but at a minimum, we would like to distinguish the following cases:

1. The file was copied.
2. The file `a` does not exist.
3. The file system was offline.
4. The program `cp` could not be loaded and run.

These distinctions are important because they guide recovery actions. In the first case, we ought to return immediately, indicating success. In the second case, the fault most likely lies with the submitter of the command, so we ought to return immediately indicating failure. In the third and fourth cases, we may wish to retry the operation several times, with a reasonable expectation that the system will repair itself.

Sadly, these distinctions are rarely available. In case of success, a program exits with code 0. The other three cases are not distinguishable. In the second case, most programs examine the arguments and then exit normally with a result of 1. The third case would appear by description to be exceptional. However, in C-based programs, such errors are detected as an ordinary failure of an I/O call, resulting in an normal exit with code 1. In Fortran-based programs, such failures result in an inability to execute an OPEN, causing an abnormal exit. The fourth case may sometimes be distinguished from the others by a failure to create the process, rather than an error in its exit code. However, failures during runtime dynamic linking also result in the indistinguishable exit code 1.

It may fairly be observed that these distinctions may be made for some programs some of the time. For example, many versions of grep, a tool for searching files, use different normal exit codes to distinguish between "file not accessible" and "no matches found". However, such programs are rare, and there exists no convention for these codes.

A similar discussion applies to many interfaces beyond local process invocation. For example, the GRAM [8] remote process invocation interface provides over 160 unique error codes detailing how GRAM itself may fail, but does not provide the exit status of complete jobs. The File Transfer Protocol [22] uses the single code 550 to represent any error at all discovered in a remote file system. The AFS [15] distributed file system uses the same error code (EACCESS) to represent both authorization failure and credential expiration. Many more examples abound. A lack of failure detail may be found in nearly every sort of programmable interface.

Consequences. It might be argued that this situation is hopeless: that such programs and interfaces are archaic and should be replaced with a more modern invocation system that hides more errors, provides more details, and gives the user better control of timing at the process level. We readily admit to making such an argument [29] relating to interpreted programs in distributed systems.

However, we live and work a world populated by these programs. Rewriting them all to correspond to a new ethic of interaction is simply not practical. Instead of simply scolding this situation, we wish to explore the possibility of constructing a reliable system from these components. In particular, we believe that the Ethernet philosophy to resource management is appropriate when both control and information are limited.

In particular, we want a language and a system that assists with the detection and handling of all failures, does not unduly clutter the program with recovery code, simplifies the use of alternate resources for recovery, and returns some control over timing to the user. These goals are similar to those raised (no pun intended) in favor of the exception [12] as a language feature for dealing with failures. We will see below that exceptions have a very different character when failure detail is unavailable.

3. The Ethernet Approach

In allocating resources, strive to avoid disaster, rather than to attain an optimum. [17]

We begin by reviewing the properties of Ethernet that are applicable to the management of computing resources other than networks. [20]

Carrier sense. An Ethernet-like client cannot consume resources at will. Before using a resource, it must wait until sufficient capacity becomes idle and then perform an acquisition protocol to allocate what it needs. On a network, this means listening for a silent period, while on a storage device, this may mean watching until sufficient space is free. The acquisition protocol is permitted to occasionally fail, allocating a resource to more than one resource. This is permitted because of the second property:

Collision detect. A client must be wary when using a newly-gained resource. It may also be in use by another client; this is known as a collision. This may be due to a race condition in the acquisition protocol, or because the underlying medium simply has underlying flaws. In order to detect collisions, the client must observe the effects of its actions rather than simply assume their success.

Exponential backoff. Collisions suggest that the short-term load on the system is greater than can be served. Because there is no central authority, clients must individually be responsible for reducing the instantaneous load by delaying and trying again with an increasing delay. Of course, the problem will not be solved if all clients return at the same instant, so some asymmetry or random factor is needed to discourage cascading collisions.

Limited allocation. Even after fairly acquiring a resource and using it without collision, a client must release it periodically to permit others to compete in the acquisition protocol. Without this requirement, other clients may be starved of any service at all.

If we are not careful to preserve all of these requirements, then we are left with a very different protocol. [6] For example, if we are unable to provide a form of carrier detect, we do not have the Ethernet protocol, but instead something like the Aloha [2] satellite network protocol. The key difference is that an Aloha-like client consumes resources at will, only detecting collisions after the fact. As we will see below, an Aloha algorithm can be implemented with less knowledge of the underlying system, but at a significant cost in performance under load. (The original Aloha network would saturate at an offered load of 18 percent.)

From these detailed properties, we may also infer some more philosophical design properties that may we borrow for software systems.

Clients are responsible for efficiency. Computer systems are full of many resources, both major and minor, that have no central authority for allocation and management. Grid computing clients must accept some responsibility for ensuring that these resources are used efficiently. They need not necessarily preserve absolute fairness, but certainly must not starve other waiting clients.

Failures may be turned into a performance problem. Given sufficiently powerful tools for resetting state and harnessing alternate resources, any minor failure may be turned into a reallocation of resources at some cost in performance. As we pointed out earlier with NFS, different users will wish to strike this balance differently.

Keep the user interface simple. If we consider both Ethernet switches and IP routers, we see that the user interface is both simple (packets go in, packets come out) and implementation independent, but the administrative interface (e.g. SNMP [7]) for debugging and tuning is both complex and implementation dependent. [1] Our approach will allow very simple language expression while allowing for the possibility of debugging and management through back channels.

4. The Fault Tolerant Shell

To explore the Ethernet approach, we introduce a scripting language called the **fault tolerant shell** or **ftsh**. The techniques we describe are possible in any language, but the introduction of a specialized syntax serves to make our examples succinct and watertight. Here, we will sketch the unique features of the language. Further details may be found in a technical report. [27]

Like any other shell, ftsh is a nested procedural scripting language whose atoms are external commands. Compound procedures may be built up by combining atoms and structural elements. A procedure, atomic or compound, does not return any value, but simply succeeds or fails. An external command succeeds if it exits normally with an exit code of zero and fails otherwise. Compound procedures also succeed or fail based on their contents. A sequence of atoms is known as a group. This group fetches an archive from a web server, uncompresses it, and then unpacks it:

```
    wget http://server/file.tar.gz
    gunzip file.tar.gz
    tar xvf file.tar
```

A group is executed sequentially and succeeds if all of its components succeed. If any component fails, the entire group fails immediately without executing the remainder. Thus, if **gunzip** above were to fail, the entire group will fail without executing **tar**.

The **try** construct is our primary tool for the Ethernet approach and is the heart of ftsh. **Try** attempts to execute a group within a given time limit. The contained group may be executed any number of times within that limit. If it succeeds, then the try construct succeeds. If the limit expires without a success, then the try expression fails. If the limit should expire during the execution of a procedure, then that procedure is forcibly terminated and the resources it consumes are freed. For example, the previous example may be attempted for 30 minutes:

```
try for 30 minutes
    wget http://server/file.tar.gz
    gunzip file.tar.gz
    tar xvf file.tar
end
```

If the contained group should fail, then the **try** delays before attempting it again. The base delay is one second, doubled after every failure, up to a maximum of one hour. Each delay interval is multiplied by a random factor between one and two in order to distribute the expected values. If the expected time of the operation is unknown, the **try** may also be expressed as a maximum number of attempts, with or without a time limit, such as `try 5 times` or `try for 1 hour or 3 times`.

The **try** may also be used to catch and react to failures in the same manner as an exception in other programming languages. The simple command **failure** is equivalent to a failed external command or the **throw** command found in other languages. No exception detail is provided to the program, as none is available in a structured way to the shell.

For example:

```
try 5 times
    wget http://server/file.tar.gz
catch
    rm -f file.tar.gz
    failure
end
```

The **forany** construct attempts to execute any single alternative of a group to success. If one succeeds, then the **forany** itself succeeds, setting the alternative variable to the successful value. For example, this fragment attempts to retrieve a file from any one of three named servers:

```
forany server in xxx yyy zzz
    wget http://${server}/file.tar.gz
end
echo "got file from ${server}"
```

As the name suggests, the **forall** construct attempts to execute all of its alternatives in parallel. If they all successfully complete, then the **forall** returns success. If any fails, all outstanding branches are aborted, and the **forall** returns failure. For example, this fragment attempts to retrieve three files from the named server.

[1] We thank Don Petravick for making this observation.

```
forall file in xxx yyy zzz
    wget http://${server}/${file}
end
```

The number of alternatives that a **forall** may execute simultaneously is of course limited by any number of local resources limits such as memory, disk space, or fixed kernel tables. Thus, the creation of processes must be governed by an Ethernet-like algorithm similar to that of **try**. We will not address this issue any further in this paper, as the behavior of **try** is enough to occupy our attention here.

Because **try** itself is a compound procedure with a result of **success** or **failure**, it may be nested, allowing for failure conditions at each component. In the following example, each attempt to retrieve the file is limited to five minutes, while the combined unpacking group is limited to one minute or three attempts, whichever expires first. The outer time limit of thirty minutes applies regardless of the depth of nesting.

```
try for 30 minutes
    try for 5 minutes
        wget http://server/file.tar.gz
    end
    try for 1 minute or 3 times
        gunzip file.tar.gz
        tar xvf file.tar
    end
end
```

Try may be placed within a **forany** or **forall** to add resilience to any one branch. It may also be placed outside in order to create retrial of the whole tree or cancellation after a time. For example, this fragment attempts to retrieve a file for 1 hour, limiting each attempt at each server to five minutes each:

```
try for 1 hour
    forany server in xxx yyy zzz
        try for 5 minutes
            wget http://${server}/file
        end
    end
end
```

It is important to note that ftsh cannot be applied blindly. Programs must be constructed with the understanding that processes will be aborted and restarted. Thus, potentially repeated actions must be idempotent. For example, the **rm** command used above is given the **-f** option to instruct it to return success if the named file does not exist.

Some abstractions require more effort. Due to the many ways that a single command may be repeated, either partially or to completion, the input and output streams of a ftsh program may become quite confused with partial results. In most shells, this problem is attacked by using external storage to hold results in abeyance, creating a simple form of I/O transaction. For example, this fragment redirects the output and error streams to and from the file `tmp`, thus pausing output until the command completes:

```
try 5 times
    run-simulation >& tmp
end
cat < tmp
```

However, this approach introduces new problems. The user must then worry about cleaning up the external storage after a failure and must also provide unique names by way of process ids or other identifiers. ftsh addresses this problem by allowing programs to redirect standard input and output to privately named variables via POSIX pipes. Such variables may be stored in the shell's memory directly, or may be kept in an appropriate place in the filesystem according to the user's or administrator's policy. Redirection to variables takes the same form as redirection to files, except that a dash prefixes the arrow operator:

```
try 5 times
    run-simulation ->& tmp
end
cat -< tmp
```

ftsh is currently implemented in POSIX C as an interpreted language in a manner similar to that of the Bourne or C shells. While executing a script, ftsh keeps a log of varying detail about the program. Online or post-mortem analysis may determine more detailed reasons for process failure, the exact resources used to execute the program, the frequency of each failure branch, and so forth.

Whenever ftsh creates a new child process, it allocates a new POSIX session id with **setsid**. POSIX allows for an entire process session to be terminated with a single system call, allowing for easy cleanup when **try** timeouts occur. Such processes are first gently requested to exit with SIGTERM and later forcibly killed with SIGKILL. Although effective in most cases, this technique has limits. A process may escape the control of ftsh by manually creating a new session id. Therefore, ftsh is appropriate as a resource management tool, but not as a security mechanism.

Exactly this problem occurs when one ftsh script executes another as an external command. In this case, the parent shell, the child shell, and the grandchildren all run in different process groups. ftsh handles this gracefully by trapping the warning **SIGTERMs** from its parent and then reacting by killing its own children. The timeout which leads to a forcible kill must be shorter in the child script; this is passed through an environment variable. This technique has worked so far in practice, but we must acknowledge that, in a heavily loaded system where the delivery of signals may be delayed, it is possible that a grandchild could escape destruction.

In other operating systems, such as Windows NT, child processes may be created within involuntary nested groupings, allowing for their reliable destruction on termination. ftsh would have a more reliable implementation on such a platform. Although the race condition is small and yet to be observed, it is unfortunate that a proper facility is not available within POSIX.

5. Applications

To demonstrate the resilience of the Ethernet approach, we will present three scenarios relevant to grid computing: job submission, disk allocation, and data transfer. In each case, we will demonstrate how system performance scales with the number of clients accessing a shared resource. We must preface these explorations with a caveat offered by several network researchers:

No real Ethernet should be operated this way. [6]

Our intention is to demonstrate resilience to resource contention. Grid computing systems will suffer frequent and unexpected bursts of contention, but this should not be considered an appropriate continuous mode of operation. Systems should be engineered with sufficient resources for production loads. The initiation of Ethernet protocols to deal with contention should be logged and noted to administrators so that persistent overloads may be accommodated.

To evaluate each scenario, we show three possible client algorithms, all implemented with minor variations on scripts written with ftsh. A **fixed** client aggressively repeats its assigned work without delay and without regard to any sort of failure. An **Aloha** client uses the ordinary ftsh **try** structure to repeat a work unit with an exponential backoff and random factor in case of failure. An **Ethernet** client uses the same structure, but additionally adds a small piece of code to perform carrier sense before accessing a resource. We will see that such small additions have a significant effect on system stability.

In each case, we will show that fixed clients scale poorly under high loads, frequently crashing to zero throughput. Aloha clients might be described as "hobble in" engineering. Although they are affected significantly by resource contention, they generally manage to maintain some level of throughput, allowing the load to be worked through. Ethernet clients maintain higher levels of throughput even under high loads by measuring the resource state and backing off before contention becomes unbearable.

Our first scenario consists of a large number of clients attempting to submit jobs into a Condor system. Each is trying to run a submitter to communicate with a Condor **schedd**. The schedd is an agent that works on behalf of a grid user, keeping jobs in a persistent queue while finding sites where they may run. A Condor submitter is a standalone executable that examines a job description file, connects to a schedd, and transfers the necessary details and files. We expect that large numbers of submitters will compete for a schedd in systems such as Chimera [10], which manage large trees of dependent tasks for a user, dispatching new jobs as old ones complete.

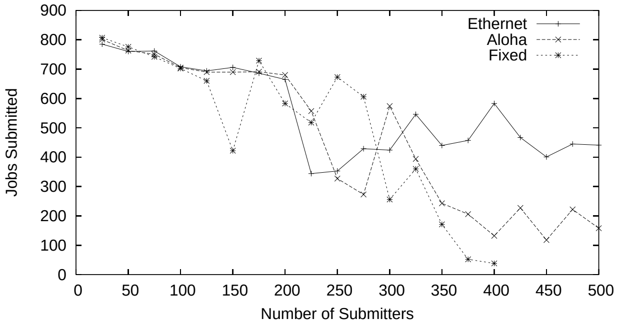

Figure 1. Scalability of Job Submission

When composing this scenario, we postulated that there would be contention for a number of expected resources: network connections, physical memory, perhaps even the disk on which the job queue is stored. In fact, it came from an unexpected source: the number of available file descriptors (FDs.) Most systems go to great lengths to manage the use of physical resources such as disks, memories, and CPUs. This overlooked resource is just as vital in a system under a heavy load.

The Aloha client in this scenario is represented by the simple program:

```
try for 5 minutes
        condor_submit submit.job
end
```

The Ethernet variant senses the "carrier" of competing clients when the number of free FDs falls below a critical value and forces the client to defer:

```
try for 5 minutes
    cut -f2 /proc/sys/fs/file-nr -> n
    if ${n} .lt. 1000
        failure
    else
        condor_submit submit.job
    end
end
```

Figure 1 shows the throughput of a varying load of submitters competing for a schedd. Each point represents the number of jobs submitted in five minutes by the given number of submitters. The fixed client fails completely above a load of 400 submitters. The Aloha client settles into an

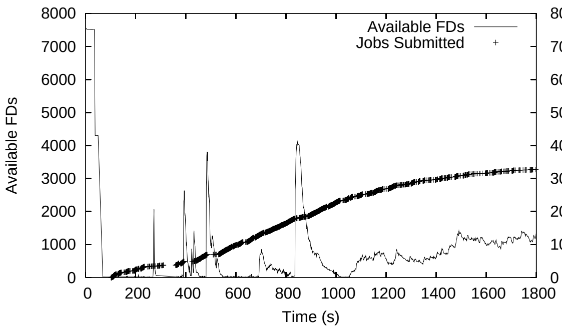

Figure 2. Timeline of Aloha Submitter

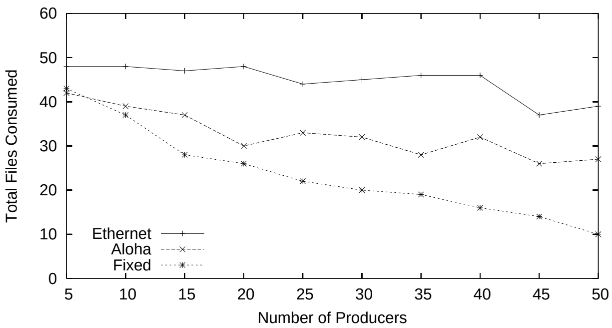

Figure 4. Buffer Throughput

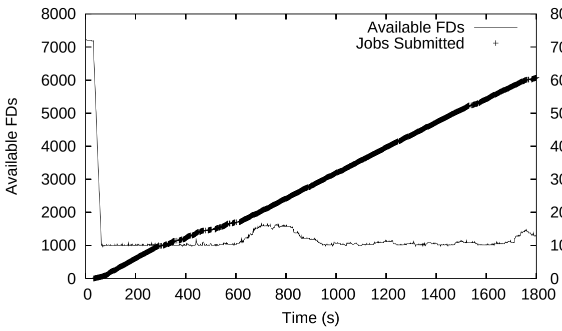

Figure 3. Timeline of Ethernet Submitter

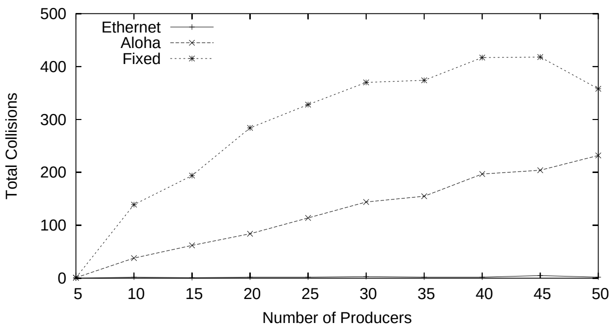

Figure 5. Buffer Collisions

unstable throughput of 100-200 jobs per five minutes, but continues to operate as load increases. The Ethernet client maintains about 50 percent of peak performance under load, due to competition for managed resources, such as the CPU.

Figures 2 and 3 clarify the reason for the throughput difference between the Aloha and Ethernet clients. Both figures show the progress of 400 clients continuously submitting jobs to a schedd over the course of thirty minutes. In each figure, the heavy dotted line shows the progressive number of jobs submitted, while the lighter line shows the number of available FDs. The Aloha clients immediately consume all of the FDs then immediately fail and backoff. The random retry factor begins to distribute the clients in time, and the consumption of FDs begins to rise to normal levels again. At several points, the number of available FDs spikes upwards. This is due to the schedd itself failing when it cannot allocate enough FDs. This, in turn, causes all of its connected clients to fail and backoff, serving as sort of a "broadcast jam" when load is extraordinarily high. The Ethernet client attempts to preserve a critical value of file descriptors. The result is that an acceptable number of clients are continually running, keeping the FDs at a high utilization.

Our second scenario is a producer-consumer problem for a shared filesystem. A number of jobs running in a remote cluster produce data whose size is not known beforehand. As they run, they place their output files into a shared filesystem buffer of 120 MB, where a consumer process collects the outputs and transmits them off to a remote archive in a manner similar to that of Kangaroo. [28]

Each producer is a continuous loop, producing an output file of random size between 0-1 MB every second. If the output cannot be written, it is deleted and a fixed, Aloha, or Ethernet retry technique is applied as above. If the output is completed, it is atomically renamed to the form `x.done` to advise the consumer that it is complete. The consumer continuously reads files at a rate of 1 MB/s, deleting each as it is consumed.

In the previous scenario, we used a "reasonable" fixed value to give the client some knowledge of when resources were running low. The problem of disk space is harder, because a client may not even know what the size of its output will be. However, the client of this scenario does have the advantage of observing the other files in the buffer, both complete and incomplete. To estimate the available disk space, the Ethernet client assumes the incomplete items in the buffer will be the same size as the average of the complete files, and subtracts that from the free disk space reported by the file system. If there is any space remaining, the client proceeds to write, otherwise it fails and backs off.

Figure 4 shows the relative throughput of each client discipline. In a manner quite similar to that of the first scenario,

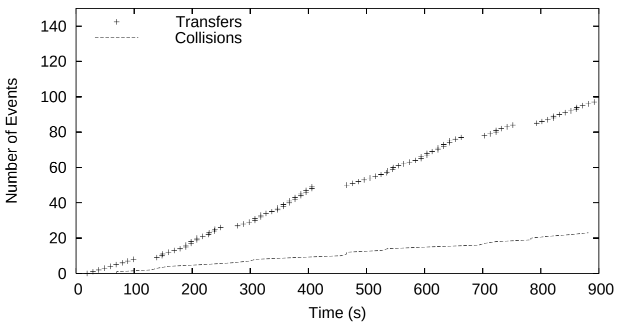

Figure 6. Aloha File Reader

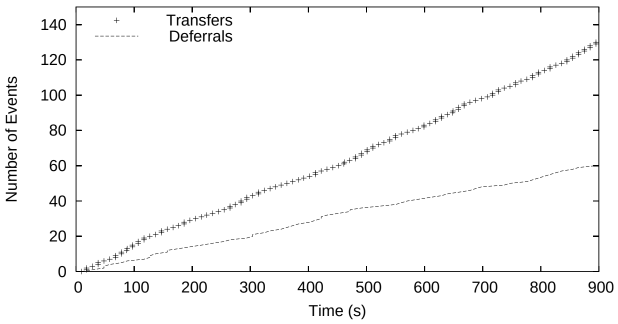

Figure 7. Ethernet File Reader

the fixed and Aloha disciplines do not scale. The Ethernet approach scales acceptably, falling off only slightly under heavy load.

The reader may question whether it is wise to design a system without a mechanism for allocating storage space independently of data transfer, such as that found in NeST [4], SRB [3], and SRM [24]. Although we certainly encourage the use of such devices, they are not deployed in every place where output data is written, nor is it clear what allocation policy would be appropriate when output sizes are not known. Further, the actual process of allocation itself may be subject to contention. We hold that any system has some unreservable resources that may be a source of contention when under heavy loads.

For our third scenario, we demonstrate a programmable solution to the problem of "black holes." Black holes are services that endlessly block or terminate any interacting client process, thus slowly bringing a system to a halt. This experimental setup consists of three web servers that replicate a read-only file service to three clients. Each client repeatedly attempts to read a 100 MB file from a server chosen at random. This takes about 10 seconds under ideal conditions. Each server is single-threaded, allowing only one client at a time to transfer data. One of the three is a permanent black hole. It permits clients to connect, but does not provide data or voluntarily disconnect.

The Aloha client in this system has a problem. It must select a timeout small enough that it will not wait unnecessarily if it should accidentally connect to a black hole. On the other hand, a timeout that is too small may accidentally abort a legitimate transfer that has been delayed for other reasons. We choose a timeout of sixty on the unsatisfactory basis that it is a good round number:

```
try for 900 seconds
    forany host in xxx yyy zzz
        try for 60 seconds
            wget http://$host/data
        end
    end
end
```

An Ethernet client attacks this problem by developing an inexpensive test to see if a server is available. It simply attempts to fetch a well-known one-byte file. If that should succeed, it proceeds to download the large file with a fair assurance that the server is working. If it should fail, the server may be a black hole or simply heavily loaded. In that case, the **forany** chooses another server. For example:

```
try for 900 seconds
    forany host in xxx yyy zzz
        try for 5 seconds
            wget http://$host/flag
        end
        try for 60 seconds
            wget http://$host/data
        end
    end
end
```

Figures 6 and 7 compare the throughput of these approaches. Predictably, the Aloha clients occasionally all fall on the single black hole server and must wait the full sixty seconds before failing and trying elsewhere. The Ethernet clients are much more effective and suffer from no such hiccups.

6. Discussion

The strength of the Ethernet approach is its wide applicability and performance robustness to a variety of situations. When the source of contention is unknown to the programmer, some level of throughput can be maintained by using simple exponential backoff to work through the burst. If more knowledge about the system is available, an appropriate "carrier sense" can be implemented to improve throughput.

We do not advocate that blindness to the cause of an error is optimal. Indeed, we would prefer to use tools that describe errors to a sufficient level of detail. But, constructing an interface with the right level of detail is difficult, and perhaps cannot be done in a way that satisfies all users. To this end, we consider the Ethernet approach to be valuable and necessary, but not ideal.

The major weakness of the Ethernet approach lies in detecting errors in the specification of a task. For example, a remotely executed job may fail because the given executable is corrupt or the arguments are simply wrong. A blind Ethernet approach may attempt such a job over and over with no hope of success. How are such situations to be handled?

The appropriate solution is to gain more information through positive activity. For example, ftsh may be used to test an executable locally on a short input file before submitting it elsewhere. Or, the same program could be attempted at multiple remote sites before it is declared a failure. Or, the presence of files named in the arguments can be tested before execution. None of these tests is an absolute guarantee, but serves to reduce the possibility of specification errors discovered at runtime.

This idea is used to great effect in Autoconf [19], a tool for configuring source code to the details of a compiler and operating system. Autoconf works by simply attempting what the user desires to do, rather than attempting to reason about it. For example, the ability to memory-map a file is probed by compiling a small program to do just that, rather than make inferences based on the system details. The former is a far more reliable and portable test.

The Ethernet approach is well known in lower level software. For example, the NFS protocol employs exponential backoff in the presence of failures, and the **wget** utility employed above has a built-in facility for retrying failed transfers. These and similar ideas have historically been used at or below the process level, perhaps in an attempt to hide from the end user the ugly realities of reliable software.

The same techniques are necessary in glue languages for several reasons. Even when the underlying tools are "clever," the costs of communication and process invocation are themselves a source of failure and resource contention before clever tools even take control. Sometimes tools may retry failures beyond the needs of the calling user, wasting both resources and time. In conventional languages, canceling such a runaway activity is quite complex, because the aborted thread may leave memory, monitors, and other resources in an unknown state. In contrast, a POSIX process is a natural unit for cancellation, because it associates a thread of control with all the resources it consumes. Memory is freed, files are released, and network connections are forcibly broken, triggering exceptions with peers. This ability to cleanly abort a running task makes the Ethernet approach uniquely suited to high-level programming.

7. Related Work

A number of methods have been proposed for dealing with failures and timeouts in general-purpose languages in a systematic way. The most widespread language structure for dealing with failures is the exception. [12] Various languages differ on critical elements of the exception concept, such as the requirement that a procedure declare all exception types that it may throw. This precise problem has driven an argument [5] against their use. ftsh side-steps this debate by using only untyped exceptions. The notion of a distinct error-value that short-circuits sequential evaluation appears in several languages. An early example is the **freturn** feature of Snobol. [13] Instead of a distinct error-value, many shell languages allow the explicit short circuiting of a group of commands with the && delimiter. This behavior is implicit in ftsh. The brittle property of ftsh bears a similarity to a special switch in the C shell which causes any failure in a sequential list to abort the entire script.

The integration of time and alternation into general-purpose languages has been less successful. Practical tools such as pdsh [11] attack the problem of running a command on many nodes of a cluster. The notion of an alternative command (like **forany**) succeeding on the completion of any of its branches is proposed by Hoare's CSP. [14] A variation is introduced by Ada, [26] which permits a default timeout in a **select** to permit real-time termination constraints. Although ftsh expresses the expiration of time as an exception which unwinds the stack, this has not been the case in most other languages. For example, in POSIX C [16], an alarm clock raises a signal, which generates a new context to handle it, rather than raising an exception in existing threads. Even with a mechanism to associate a timeout with a running thread, the forcible cancellation of threads in a running language is difficult or impossible, because a thread runtime does not associate program resources with the thread that allocated them.

8. Conclusion

> *I would therefore like to posit that computing's central challenge, "How not to make a mess of it," has* not *been met. [9]*

Current grid computing systems are so complex as to defy the ability of even specialists to deploy and use them without going to extraordinary lengths to tune and debug. We humbly admit to contributing a fair share of the "mess." These systems are so hard to use in part because they are sensitive to an extraordinary set of unexpected failure modes.

Our contribution is the proposition that failures should not be hidden in the depths of a system. Rather, the likelihood of failure and the mechanisms for fault tolerance should be expressed at the highest levels of programming in simple terms such as retry and alternation. This is necessary because both users and designers assemble systems out of disparate components that cannot be expected to choose the right remediation strategy among themselves.

Just as a single obnoxious customer can disrupt a movie theater, any misbehaved client can ruin the Ethernet approach. If the clients of a service cannot be trusted to play fairly, then the only solution is to physically isolate them. This could be accomplished to a certain extent if systems allowed guaranteed allocations for all resources. However, even such systems have some uncontrolled, shared resources: the entry point for requests. Whether it is a ticket window at a movie theater or a TCP port for a web server, any system has some resource that must be consumed cooperatively. The Ethernet approach is needed wherever such resources are found.

Drawing on existing languages, we have proposed a simple language to allow the user to express fault tolerance – literally, the user's limit of tolerance for failures – in a simple and concise manner. The Ethernet approach to distributed computing seeks to avoid disasters while providing acceptable performance with a minimum of fuss in a wide variety of situations.

Further information about ftsh may be found at `http://www.cs.wisc.edu/condor/ftsh`.

9. Acknowledgments

We gratefully acknowledge fruitful conversations with many members of the Condor team, including Peter Keller and Marvin Solomon. This work was supported in part by a Lawrence Landweber NCR Fellowship and the Wisconsin Alumni Research Foundation.

References

[1] H. Abelson, G. Sussman, and J. Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Massachusetts, 1996.

[2] N. Abramson. The ALOHA system - another alternative for computer communication. In *Proceedings of the Fall Joint Computer Conference*, pages 281–285, 1970.

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

[4] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. A. Dusseau, R. Arpaci-Dusseau, and M. 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.

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

[6] D. Boggs, J. Mogul, and C. Kent. Measured capacity of an ethernet: Myths and reality. Technical Report Research Report 88/4, Western Research Laboratory, September 1988.

[7] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol. Request For Comments 1157, Internet Engineering Task Force, 1990.

[8] 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.

[9] E. Dijkstra. The end of computer science? *Communications of the ACM*, 44(3):92, March 2001.

[10] I. Foster, J. Voeckler, M. Wilde, and Y. Zhou. Chimera: A virtual data system for representing, querying, and automating data derivation. In *Proceedings of the 14th Conference on Scientific and Statistical Database Management*, Edinburgh, Scotland, July 2002.

[11] J. Garlick. The parallel distributed shell. http://www.llnl.gov/linux/pdsh/pdsh.html.

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

[13] R. E. Griswold, J. F. Poage, and I. P. Polonsky. *The SNOBOL4 Programming Language*. Prentice Hall, Englewood Cliffs, New Jersey, 1971.

[14] C. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[15] 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 Systmes*, 6(1):51–81, February 1988.

[16] IEEE/ANSI. Portable operating system interface (POSIX): Part 1, system application program interface (API): C language, 1990.

[17] B. W. Lampson. Hints for computer system design. In *Proceedings of the 9th ACM Symposium on Operating Systems Principles*, volume 17, pages 33–48, 1983.

[18] D. D. E. Long, A. Muir, and R. A. Golding. A longitudinal survey of internet host reliability. In *Symposium on Reliable Distributed Systems*, pages 2–9, 1995.

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

[20] R. Metcalfe and D. Boggs. Ethernet: Distributed packet-switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[21] 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.

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

[23] 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.

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

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

[26] S. Taft and T. Duff. *Ada 95 Reference Manual*, volume 1246 of *Lecture Notes in Computer Science*. Springer Verlag, 1995.

[27] D. Thain. The fault tolerant cookbook. Technical Report UW-CS-TR-1476, University of Wisconsin, Computer Sciences Department, 2003.

[28] D. Thain, J. Basney, S.-C. Son, and M. Livny. The Kangaroo approach to data movement on the grid. In *Proceedings of the Tenth IEEE Symposium on High Performance Distributed Computing (HPDC10)*, San Francisco, California, August 2001.

[29] D. Thain and M. Livny. Error scope on a computational grid. In *Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing (HPDC)*, July 2002.