Phoenix: Making Data-intensive Grid Applications Fault-tolerant

George Kola, Tevfik Kosar and Miron Livny
Computer Sciences Department, University of Wisconsin-Madison
1210 West Dayton Street, Madison WI 53706
{kola,kosart,miron}@cs.wisc.edu

Abstract

A major hurdle facing data intensive grid applications is the appropriate handling of failures that occur in the grid-environment. Implementing the fault-tolerance transparently at the grid-middleware level would make different data intensive applications fault-tolerant without each having to pay a separate cost and reduce the time to grid-based solution for many scientific problems. We analyzed the failures encountered by four real-life production data intensive applications: NCSA image processing pipeline, WCER video processing pipeline, US-CMS pipeline and BMRB BLAST pipeline. Taking the result of the analysis into account, we have designed and implemented Phoenix, a transparent middleware-level fault-tolerance layer that detects failures early, classifies failures into transient and permanent and appropriately handles the transient failures. We applied our fault-tolerance layer to a prototype of the NCSA image processing pipeline and considerably improved the failure handling and report on the insights gained in the process.

1. Introduction

A major hurdle facing data intensive grid applications is appropriate handling of failures that occur in the grid-environment. Most application developers are unaware of the different types of failures that may occur in the grid environment. Understanding and handling failures imposes an undue burden on the application developer already burdened with the development of their complex distributed application.

We feel that grid middleware should tolerate grid faults and make this functionality transparent to the application. This would enable different data-intensive grid applications to become fault-tolerant without each having to pay a separate cost. Removing the burden of understanding and handling failures lets application developers concentrate on the problem at hand and reduces the time to grid-based solution for their problem.

We have developed Phoenix, a transparent grid middleware solution that adds fault-tolerance to data intensive grid application by detecting failures early, classifying failures into transient and permanent, and handling each transient failure appropriately.

2. Background

In this section, we give the widely accepted definitions for faults, errors and failures, discuss the issue of error propagation, and compare our approach to existing related work.

Faults, Errors and Failures

The widely accepted definition, given by Avizienis and Laprie [3], is as follows. 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.

A fault need not result in an error nor an error in a failure. An alpha particle corrupting an unused area of memory is an example of a fault that does not result in an error. In the Ethernet link layer of the network stack, a packet collision is an error that does not result in a failure because the Ethernet layer handles it transparently.

Error Propagation

In a multi-layered distributed system where layers are developed autonomously, what errors to propagate and what errors to handle at each level is not well understood [20]. The end-to-end argument [19] states that the right place for a functionality is the end-point, but that it may be additionally placed in the lower levels for performance reasons.

Pushing all the functionality to the end-point increases its complexity and requires the end-point developers to understand all errors that may occur in the underlying layers.

In a grid environment, where application developers are domain experts and not necessarily grid experts, requiring application developers to understand grid errors would mean that they might never complete their application.

An alternate approach followed in many multi-layered systems including the network stack is to make each layer handle whatever error it can and pass up the rest. This masking of errors, while reducing higher-level complexity, hurts the performance of sophisticated higher layers that can use this error information to adapt.

Thain and Livny [20] have developed a theory of error propagation. They define error scope as the portion of the system an error invalidates and state that an error must be propagated to the program that manages its scope. Applying their theory, we find that grid errors are of grid-middleware scope and not necessarily of application scope. Therefore, a grid middleware layer may handle most grid errors. Aided by this theory, we decided to add fault-tolerance capability at the grid-middleware level. To handle the information loss and to enable sophisticated applications and allow interposition of adaptation layers between our middleware layer and the application, we persistently log the errors encountered and allow tuning of the error masking. Logging of the errors helps performance tuning and optimization.

Related Work

Medeiros et al. [16] did a survey of failures in the grid environment and found that 76% of the grid deployers had run into configuration problems and 48% had encountered middleware failures. 71% reported that the major difficulty was diagnosing the failure. Our work looks at only data intensive grid applications and our framework would help in diagnosis of failures.

Hwang and Kesselman [11] propose a flexible framework for fault tolerance in the grid consisting of a generic failure detection service and a flexible failure-handling framework. The failure detection service uses three notification generators: heartbeat monitor, generic grid server and the task itself. The failure-handling framework handles failures at task level using retries, replication and checkpointing and at workflow level using alternative task, workflow-level redundancy and user-defined exception handling. Their work looks at only computation and does not deal with data placement. While heartbeats may help deal with machines that accept jobs and do nothing, they would have difficultly dealing with compute nodes in a private network and compute nodes behind a firewall. Their work does not distinguish between transient and permanent failures and a loss of heartbeat may mean a network outage(transient) or a node crash (permanent). Our work classifies failures into transient and permanent and handles each class appropriately and it takes into account the failures that can occur in data placement in addition to computational failures.

Gray classified computer bugs into Bohrbugs and Heisenbugs [9]. Bohrbug is a permanent bug whereas Heisenbug is a transient bug that may go away on retry. Our classification of failures into permanent and transient is similar to Gray's classification of bugs.

3. Why do Data Intensive Grid Applications Fail ?

We looked into the different types of failures experienced by data intensive grid applications by analyzing logs of four real-life production data intensive applications: NCSA Image processing pipeline [17], WCER video pipeline [14],US-CMS pipeline [5] and BMRB BLAST [1, 4] pipeline.

The failures in the order of frequency of occurrence

1. Intermittent wide-area network outages.
2. Data transfers hanging indefinitely. Loss of acknowledgment during third party ftp transfers was the main culprit.
3. Outages caused by machine crashes and downtime for hardware/software upgrades and bug fixes.
4. Misbehaving machines caused by misconfiguration and/or buggy software.
5. Data corruption. It happened mostly during data transfer and at times due to faulty hardware in the data stage and compute machines.
6. Insufficient disk space for staging-in input file or for writing output file.
7. Trashing of storage server and subsequent timeout due to too many concurrent read data transfers.
8. Storage Server crash due to too many concurrent write data transfers.

Failures 1,2,7 and 8 affect mostly data transfers while 3,4,5 and 6 affect both data transfers and computation. In normal grid applications, computation and data transfer failures are coupled. Thus, a data transfer failure results in recomputation. All the four applications realized that to ensure forward progress, data transfers failures should be separated from computational failures and they accomplish this by decoupling computation and data transfer.

The first three applications treat data transfer as a full-fledged job and schedule it with Stork [15], a specialized data placement scheduler. The BLAST pipeline wraps the data transfer in a fault-tolerant shell [21]. Doing the above does not completely solve the problem. The reason

Source URL (protocol://host:port/file)	Destination URL (protocol://host:port/file)	Error Code (Optional)
gsiftp://quest2.ncsa.uiuc.edu:4050/1/data/1.dat	gsiftp://beak.cs.wisc.edu:3040/tmp/1.dat	1

Table 1. Information fed to failure agent.

is that Stork following the principle of exokernel [12] implements only mechanisms and expects users and/or higher-level planners to specify policies. The policy specifies time-outs after which stork should tag a transfer as hung, number of retries etc. Ftsh, which is data placement agnostic, has difficultly cleaning up partially transferred files before retrying a data transfer and it expects threshold for retries. These timeouts and thresholds indirectly help the system classify failures into transient and permanent by stating that if the problem does not go away within this threshold of retries, it is a permanent failure.

Grid users may not be able to come up with these thresholds as they depend on end-system, network characteristics and the type of error, things not known at job submission time. Many grid deployers want the system to handle the transient failures and notify them about the permanent failures. They prefer to specify things at a higher level like try to complete the jobs as quickly as possible using up to X% extra resources. Expecting them to specify the time-outs and thresholds makes their job difficult and they usually end up choosing sub-optimal values.

In the Grid, there would always be some misconfigured/misbehaving machines and these machines may accept the job and not do anything or be unable to communicate the result or even worse they may do this for only certain classes of jobs. The middleware needs to detect such machines during job execution, treat them as resource failure, and retry the jobs elsewhere.

4. Achieving Fault-tolerance

We want to achieve fault-tolerance so that only application errors result in user perceivable failures while we transparently handle grid failures. To achieve this goal, we need to be able to first detect failures, then distinguish between grid failures and application failures, and finally handle the different grid failures appropriately.

4.1. Strategy for Detecting Failures

Hung transfers that appear like normal transfers and misconfigured machines that accept jobs and do not return any result make failure-detection difficult. To handle these, we use the gridknowledgebase [13] and extract the history of transfers/computation and fit a distribution to the transfer/compute times. Depending on the user specified threshold of false positives(e.g. 1%, 5%, etc), we set the time-limit for a transfer/computation to be mean + x(standard-deviation), where x is derived from the false-positive threshold and the distribution fitted. If no history exists, we do not set any threshold for the computation, but set a threshold for the data transfer based on a minimal data transfer rate. If the transfer/computation exceeds the threshold, the failure-detector stops it and marks it as a failure.

Transfers and computation may also fail with an error code and they are easy to detect.

4.2. Classifying Failures

After identifying failures, we need to classify them. In the computation side, researchers have done most of the work to differentiate between resource failure and application failure. Most interfaces report if the job failure was due to middleware/resource error [8]. New problems encountered when jobs run inside a virtual machine like Java Virtual Machine have been handled by having a wrapper around the job to correctly propagate the error [20].

On the data placement side, the issue of separating grid errors from application errors has not been addressed. It is made difficult by the lack of feedback from the underlying system and at times, even the underlying system may not know how to classify the error. For instance, if the source host is unreachable, either it may mean there is a network outage (grid error) or that there was a typo in source host in the data transfer job (application error).

To classify data placement failures, we propose a failure agent that takes the complete source and destination URL and optionally the return code, as shown in Table 1, and identifies the cause of the failure and interacts with the policy manager and classifies the failure into transient or permanent and gives a detailed error status.

Failure Agent

As shown in figure 1, the failure agent identifies the source of failure as follows. The failure agent checks if the appropriate DNS server is up.

Next, the failure agents checks if the source and destination have a valid DNS entry. If any does not, it is likely that the user made a typo.

If the DNS entry exists, it tries to see if that host network is reachable. If that network is not accessible, it logs it as wide-area network failure. As the failure agent may be running on a different network from the source or desti-

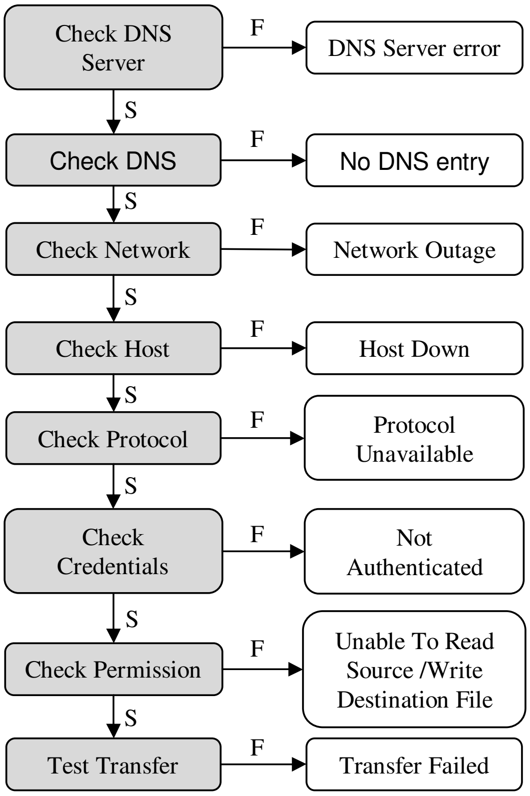

Figure 1. Shows how the failure agent detects the cause of data transfer failure.

nation, a potential problem arises if the wide area connectivity of the node running the failure agent is down. Then, the failure agent cannot work properly. We address this issue by running the failure agent on the control node for the whole processing. The control node is responsible for submitting the computation and triggering the wide-area data transfers via third party transfers. If connectivity of control node is down, it cannot start transfers and our scheme works well by marking that as wide-area network outage.

If the host network is reachable, failure agent tries to check if the host is up. If the host is down, it reports that.

After reaching the host, it tests if that protocol is available on that host. If all that works fine, for both hosts, it could be some problem with credential or some misconfiguration. The failure agent tries to authenticate that user to the system and sees if that goes fine. If it fails, it is an authentication problem.

After authentication, it checks for the access to source file and ability to write to destination file. If any fail, it logs the appropriate error. If the system gives enough feedback, the failure agent tries to differentiate between source file not existing and lack of permission to access the source file. Similarly, for destination file, it tries to distinguish between being unable to create the destination file, lack of permission to write to destination file and being unable to write any data to the destination file. The next step may be to try to transfer some dummy data to see if the server works. Optionally, this part can use a test suite that can test a data transfer service.

Authenticating the user, checking permission and running test transfers requires that the failure agent has the user credentials and the failure agent handles this by interacting with the data placement scheduler.

The user can specify policies that influence the working of the failure agent. For instance, users can specify the order of preference for methods to probe if the host is reachable. At times, users may have an unconventional setup that may confuse the failure agent and the policy mechanism allows sophisticated users to tune it to handle those cases. An example is a packet filter that is set to drop all probe packets.

4.3. Failure Manager

The failure manager is responsible for coming up with a strategy to handle transient failures. Users can influence these decisions that failure manager makes by specifying policies. Many storage systems have maintenance windows and the user policy can specify that. For instance, if the user specifies that the storage server has a maintenance window every Sunday between 4 a.m. and 8 a.m., then if the storage server is unreachable during that time, the failure manager would retry the transfer after the maintenance window. Further, some users want the system administrator notified via email if a node is down and they can specify that in the policy. Some users may have hard limits for certain jobs i.e. they want the job completed within a time limit and they do not want the system to execute that job after the time limit. The user may specify this as a policy. Users can also tune the exponential back off strategy and can even explicitly state a strategy for the different transient failures. If users do not want to specify the policy, they can tune the provided policy to their preference.

If the failure manager stores information about previous failures, it can use it to adapt its strategy on the fly. For instance, the first strategy chosen by the failure manager may not be good enough if the failure occurs again and using history, it can find out the strategies that worked well and those that did not and use it to refine future strategies. Since maintaining this state in a persistent manner and recovering from crashes considerably increases the complexity of the failure manager, we have enhanced the grid knowledgebase to store the failure information. An advantage of this is that different failure managers can share the knowledge about failure enabling each to make better decision. Keeping the failure manager stateless simplifies its design and makes crash recovery simple.

For permanent failures, we need to either consult a higher-level planner or pass the failure to the application.

4.4. Checking Data Integrity

Even though the data transfer may have completed successfully, the transferred file may have been corrupted. The only way of verifying this is through end-to-end checksum, i.e. compute source checksum(if it is not already available) and destination checksum and verify that they both match. If we cannot run computation on the destination storage server, we may need to download the written data to a nearby node and compute checksum on it and use that as the destination checksum.

As there is a significant cost associated with checksum, some users may not want to perform checksum on all the data. To help them, we have a data integrity manager that allows users to specify preference on the percentage of data they are willing to checksum. The data integrity manager turns on checksum for certain of the transfers and does this in a statistically unbiased manner.

Whenever a transfer fails a checksum, the data integrity manager figures out the cause of the data corruption and takes a suitable action based on user specified policy. For instance, if a compute node caused the data corruption, a conservative policy may be to recompute all data generated since the previously verified checksum. Another policy may be to try to do a binary search, by recomputing the results at different points and comparing the checksum of the result with that from the corrupted node. This may help us get a smaller window where the node started corrupting the data. It also depends to certain extent on the type of failure. Both the policies may not work if the node corrupts only some of the computation data, with the conservative being better.

The integrity manager can send out email to the appropriate party about the source of data corruption. It can also feed the information to the job policy manager to avoid the repeat of the problem. For instance, if a computation node corrupts data, it will make sure that jobs do not run on that node again until it is fixed.

For users who do not want to perform checksums but want to verify that all of the data has been transferred, we provide an option that verifies that the source and destination file sizes are same in addition to checking that success is returned by the protocol. We did this when we encountered protocol bugs with certain protocol that return success when only a part of the data has been transferred. This occurred in SRB protocol when the destination disk got full. Users can use this as an optimization before checking checksum, as the system does not have to compute the

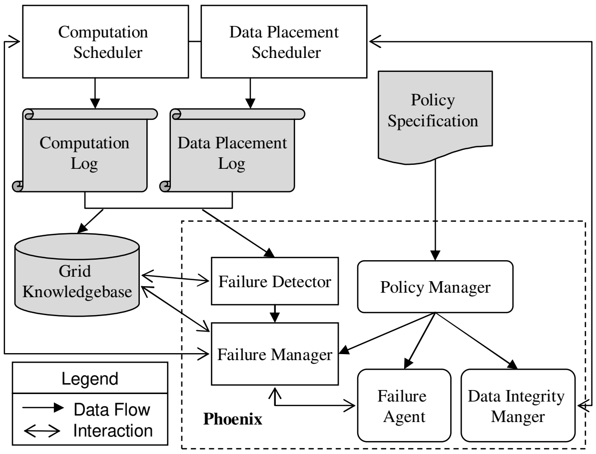

Figure 2. Shows how the different components fit together.

checksum when the source and destination file sizes do not match.

4.5. Phoenix: Putting the Pieces Together

Figure 2 shows the overview of how the different pieces of fault-tolerant middleware fit together.

The failure detector scans the user log files of computation scheduler and data placement scheduler to detect failures. It interacts with the gridknowledgebase to detect hung transfers and run-away computation. After detecting failures, it passes that information to the failure manager. For data placement failures, the failure manager consults the failure agent to find out the cause of the failure. The failure agent identifies the cause and classifies the data placement failures taking into account user specified policies acquired from the policy manager. The failure manager consults the policy manager and comes up with a strategy to handle the transient failures. It also logs the failure status information to grid knowledgebase to share that information with other failure managers and to build history to adapt itself.

The data integrity manager based on user policy turns on file size verification and checksum computation and verification for a certain percentage of the data transfers. When a transfer fails the file size verification or checksum verification, it interacts with the data placement scheduler to retransfer that data.

The failure detector, failure manager, failure agent, policy manager and data integrity together constitute Phoenix, our fault-tolerant middleware layer.

Logging the failures and the strategy taken lets users know the failures encountered. This is useful to address the

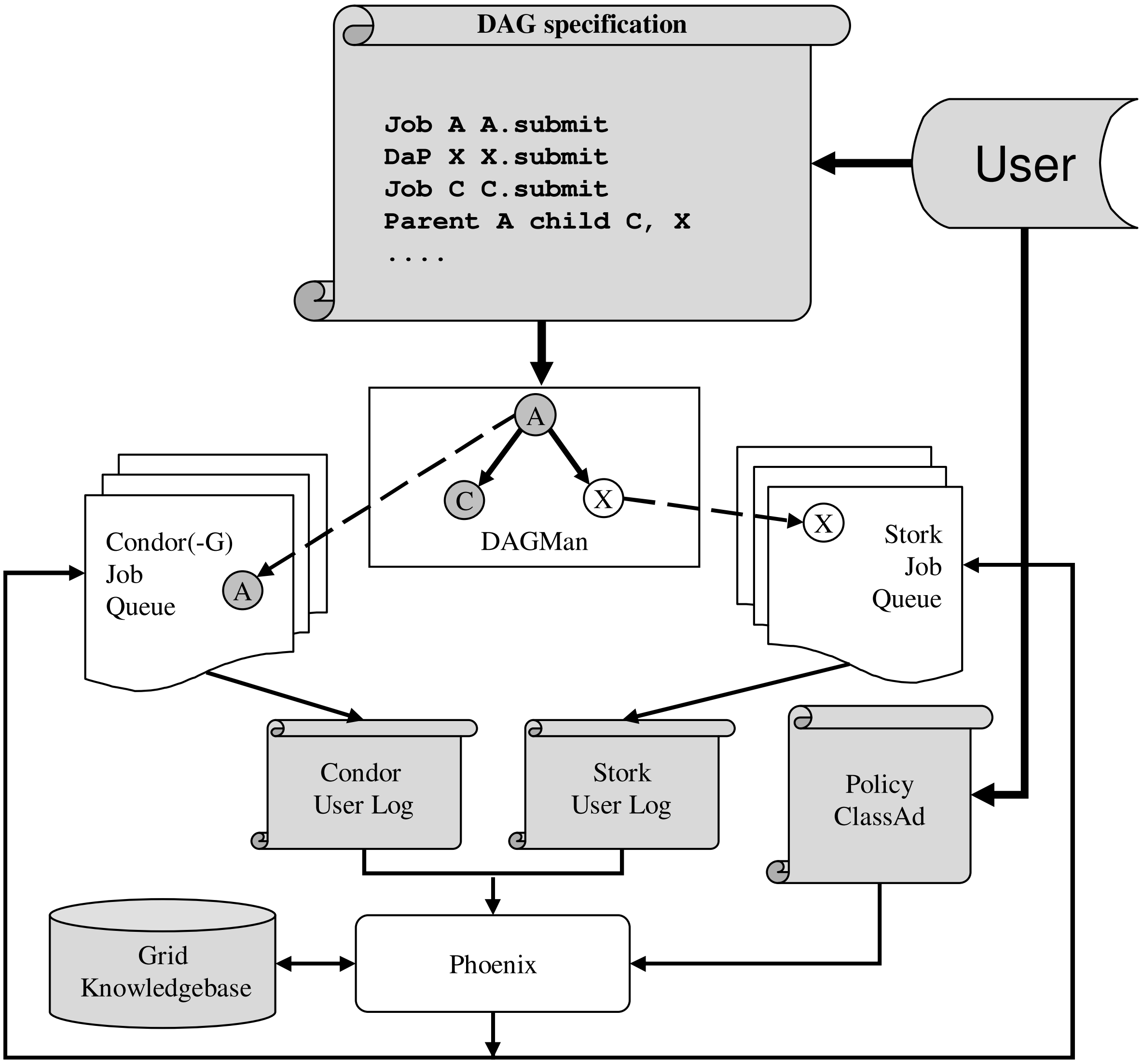

Figure 3. Framework showing how the different components fit together

earlier mentioned information loss when lower layers handle the faults encountered. So, a sophisticated higher layer, either the application or a smart layer between Phoenix and application can use this information to tweak policies of Phoenix and may even convert Phoenix policies into mechanisms by applying the infokernel [2] approach.

5. Framework

While the components we designed and implemented can be easily integrating into existing systems, we found that many users wanted a full-system solution. To address this we designed and implemented a framework that integrates our fault tolerance components. Our system uses Condor/Condor-G as the computation scheduler. As most of the grid users used Condor-G to submit their grid jobs to Grid2003 [10], they can easily employ our system and benefit from it.

The user submits a DAG specifying the different jobs and the dependencies between jobs to DAGMan and specifies the policy in ClassAd format [6, 7, 18]. DAGMan submits the computation jobs to Condor/Condor-G and data placement jobs to Stork.

Phoenix keeps monitoring Condor and Stork user log files to detect failures. It uses the gridknowledgebase to extract the history and uses it to detect hung transfers and computation.

Taking into account user specified policies, Phoenix classifies failures into transient and permanent and comes up with a suitable strategy to handle transient failures.

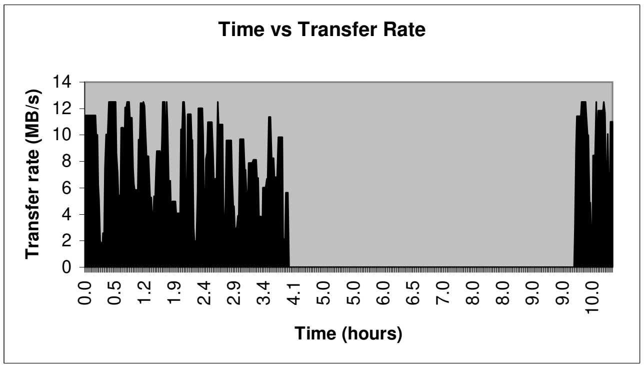

Figure 4. Shows how the failure agent detects the cause of data transfer failure.

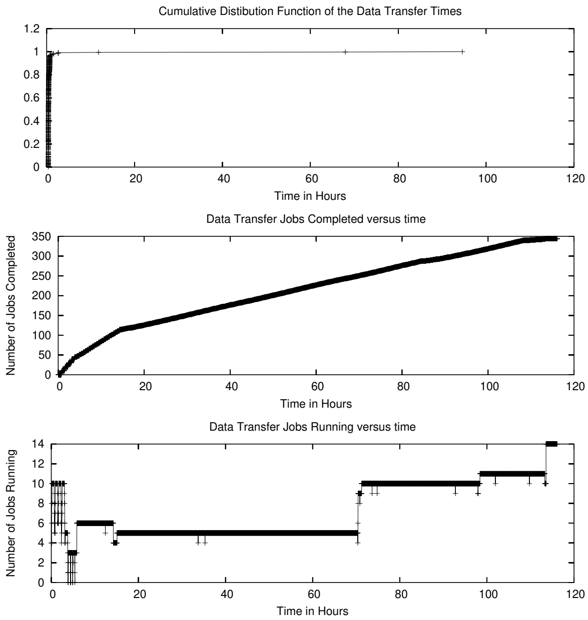

Figure 5. Shows the case of hung data transfers. Without Phoenix, users had to manually detect such failures

It logs both the failure and the strategy taken to the gridknowledgebase. Logging the failure allows users to query the gridknowledgebase for encountered failures. Phoenix can potentially use this to adapt its strategy on the fly.

Phoenix can also turn on checksum and file size verification in a statistically unbiased manner for the specified percentage of transfers. In the current form, it does not support checksum if the destination file server does not allow checksum computation to be performed. The difficulty is to come up with a suitable host to transfer the data and verify the checksum and to detect, in a low overhead manner, if this host is corrupting the data.

At present, Phoenix passes permanent failures to the application.

6. Insights from NCSA Image processing pipeline prototype

The NCSA image processing pipeline prototype involved moving 2611 1.1 GB files(around 3 terabytes) data from SRB mass storage system at San Diego Super Computing Center, CA to NCSA mass storage system at Urbana-Champagne, IL and then processing the images using the compute resources at NCSA, Starlight Chicago and UW-Madison.

During the processing, there was an SRB maintenance window of close to 6 hours. The figure 4 shows the pipeline recovering from this transient failure.

Figure 5 gives information about 354 data transfers each transferring a different 1.1 GB file with 10 transfers proceeding concurrently. It shows the cumulative distribution of the data transfer times and the number of jobs executing concurrently and the number of jobs completed over a 120-hour period.

Ignoring the outliers, most of the transfers take less than 30 minutes with a standard deviation of 9 minutes. Of the 6 outliers, 3 outliers take 2 1/2 hours each and other three vary between 12 to 96 hours. A simple kill and restart of the outlier data transfer would have resulted in those transfer completing much earlier. Such variations happen because of Heisenbugs and using Phoenix, we can detect them early and retry them to success considerably improving the overall throughput.

7. Future Work

We plan to integrate the fault tolerance layer with the production NCSA image processing pipeline and WCER video pipeline. We are also considering interfacing our system with higher-level planners like Pegasus [**?**, **?**].

8. Conclusions

We have successfully designed and implemented Phoenix, a fault tolerant middleware layer that transparently makes data intensive grid applications fault-tolerant. Its unique feature includes detecting hung transfers and misbehaving machines, classifying failures into permanent, transient, and coming up with suitable strategy taking into account user specified policy to handle transient failures. It also handles information loss problem associated with building error handling in lower layers by persistently logging failures to gridknowledgebase and allowing sophisticated application to use this information to tune it. Using a prototype of a real life NCSA pipeline, we show the usefulness of Phoenix's features.

9. Acknowledgments

We thank Robert Brunner and his group at NCSA for collaborating with us on the NCSA image pipeline, Chris Thorn and his group at WCER for collaborating on the WCER video pipeline, Zach Miller for helping us with BMRB BLAST pipeline, Dan Bradley for helping us with US-CMS pipeline and other members of the condor team for the support, comments and feedback.

References

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

[2] A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, N. C. Burnett, T. E. Denehy, T. J. Engle, H. S. Gunawi, J. A. Nugent, and F. I. Popovici. Transforming policies into mechanisms with infokernel. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 90–105. ACM Press, 2003.

[3] A. Avizienis and J. Laprie. Dependable computing: From concepts to design diversity. In *Proceeding of the IEEE*, volume 74, pages 629–638, May 1986.

[4] BioMagResBank. A repository for data from nmr spectroscopy on proteins, peptides and nucleic acids. http://www.bmrb.wisc.edu/index.html, 2004.

[5] CMS. The US Compact Muon Solenoid Project. http://uscms.fnal.gov/.

[6] N. Coleman, R. Raman, M. Livny, and M. Solomon. Distributed policy management and comprehension with classified advertisements. Technical Report UW-CS-TR-1481, University of Wisconsin - Madison Computer Sciences Department, April 2003.

[7] Condor. Classified advertisements. http://www.cs.wisc.edu/condor/classad/, 2004.

[8] J. Frey, T. Tannenbaum, I. Foster, and S. Tuecke. Condor-G: A Computation Management Agent for Multi-Institutional Grids. In *Tenth IEEE Symp. on High Performance Distributed Computing*, San Francisco, CA, August 2001.

[9] J. Gray. Why Do Computers Stop and What Can Be Done About Them? Technical Report TR-85.7, Tandem, June 1985.

[10] Grid3. An application grid laboratory for science. http://www.ivdgl.org/grid2003/, 2004.

[11] S. Hwang and C. Kesselman. A Flexible Framework for Fault Tolerance in the Grid. *Journal of Grid Computing*, 2004.

[12] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pages 52–65, Saint-Malô, France, October 1997.

[13] G. Kola, T. Kosar, and M. Livny. Client-centric grid knowledgebase. In *To Appear in the Proceedings of the 2004 IEEE International Conference on Cluster Computing (Cluster 2004)*, September 2004.

[14] G. Kola, T. Kosar, and M. Livny. A fully automated fault-tolerant system for distributed video processing and off-site replication. In *Proceedings of the 14th ACM International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV 2004)*, Kinsale, Ireland, June 2004.

[15] T. Kosar and M. Livny. Stork: Making Data Placement a First Class Citizen in the Grid. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS 2004)*, Tokyo, Japan, March 2004.

[16] R. Medeiros, W. Cirne, F. Brasileiro, and J. Sauvé. Faults in grids:why are they so bad and what can be done about it? In *Proceedings of the Fourth International Workshop on Grid Computing*, Phoenix, Arizona, November 2003.

[17] NCSA. NCSA laboratory for cosmological data mining. http://www.ncsa.uiuc.edu/AboutUs/People/Divisions/divisions54.html, 2004.

[18] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC7)*, Chicago, Illinois, July 1998.

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

[20] D. Thain and M. Livny. Error scope on a computational grid: Theory and practice. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[21] D. Thain and M. Livny. The ethernet approach to grid computing. In *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*, Seattle, WA, June 2003.