Recovery-Oriented Computing in Distributed Systems

Vincent Borchardt
Department of Computer Science
University of Minnesota, Morris
borch135@morris.umn.edu

ABSTRACT

Our computing lives are moving off the computers we own to distributed systems we access through the internet, popularly known as the "cloud". However, since we do not have direct access to the computers in the cloud, we need to trust the system, and part of that trust includes understanding how those systems prevent and recover from failures, known collectively as the systems' fault tolerance. Recovery Oriented Computing is a different approach to fault tolerance that accepts that errors will happen and instead focuses on the recovery time of the system. The paper discusses two different types of distributed systems, grid computing and stream computing, each using concepts from Recovery Oriented Computing to improve fault tolerance.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Fault tolerance, Reliability, availability, and serviceability; C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design, Performance, Reliability

Keywords

Stream computing, Grid computing, Cloud, Recovery Oriented Computing

1. INTRODUCTION

Our computing lives, from our data to the majority of our processing power, are generally moving off the computers we sit in front of to computers we access through the Internet. These computers, which are referred to as the "cloud", are part of very large systems, and the user only sees a small part of that larger whole. This larger system is known as a distributed system, and is made of many individual computers which communicate in various ways.

When the user does not have direct access to the system that is doing most of their necessary work, they have to blindly rely on the system, since it cannot be directly fixed by the user. However, in direct contrast to this blind reliance, the parts of the system can each fail in many ways, including individual failure of each part as well as failure in communication between parts. Traditionally, protection against faults in a system has been focused on stopping faults from happening completely [3]. However, an easier and more efficient way to protect against faults is to focus on returning the system to normal after a fault occurs, and this is the defining idea of Recovery-Oriented Computing [3].

This paper will focus on two different types of distributed systems: one based on grid computing and one based on stream computing. The grid computing systems we will discuss in Section 3 operate under strict time limits, and the fault tolerance of those systems come from ensuring the use of reliable sub-systems, as well as a hybrid technique of saving partial progress and duplicating other parts of the system. The stream computing system discussed in Section 4 gathers information from various sources, and communicates that information from different parts of the system in a fault-tolerant way.

2. BACKGROUND

2.1 Fault Tolerance

When building a large system that includes unreliable components, making the overall system reliable is difficult. The process of detecting errors and doing useful things when errors are detected is called fault tolerance [5].

The two main metrics used when considering if a system is fault-tolerant are the mean time to failure (MTTF) of the system and the mean time to repair (MTTR) of the system [5]. The MTTF of a system is the average time until the system fails in some way over a large number of cycles of run-fail-repair. Similarly, the MTTR of a system is the average time it takes to repair the system after a failure over a large number of run-fail-repair cycles. The availability, or uptime of the system is the total time the system was running during the cycle (MTTF), divided by the total time of the cycle (MTTF + MTTR), which gives a percentage of the time the system was up. The downtime is the time the system was not available as a percentage of the entire cycle time, which is the MTTR divided by the MTTF + MTTR.

Source of Failure	Percentage
Operator Error	51%
Hardware Error	15%
Software Error	34%

Table 1: Sources of website failures from [3]. Includes only failures with a visible cause.

2.2 Recovery Oriented Computing

The ultimate goal for a system is to have an uptime of 100%, but this requires either a system that never fails or a system that takes no time to recover, both of which are impossible. Still, the difference between 99% and 99.999% uptime can be millions of dollars, since one percent of a year is 80 hours, and the cost per hour of downtime for a large service can range from $200,000 for an internet service to $6,000,000 for a stock brokerage [3]. Using the extreme example of a stock brokerage, increasing the uptime from 99% to 99.999% would decrease the downtime from 80 hours to less than 5 minutes, reducing the cost by nearly six orders of magnitude.

There are two ways to increase uptime: increasing the MTTF or decreasing the MTTR. Since most systems are built with the mindset of completely eliminating errors, they already have a high MTTF. However, errors will happen in systems, no matter how much you try to mitigate them. To further that point, Table 1 shows what percentage of failures from three websites were caused by operator error (the person maintaining the system makes a mistake), hardware error (a failure in the physical components of the system), and software error (the programming of the system has a mistake), and over half of the errors were caused by the operators of the system [3]. In those cases, the MTTF determined from the hardware and software used by the system is much less important.

Since the MTTF of most systems is already high, decreasing MTTR is a much easier way to increase uptime in most systems. Decreasing the MTTR to improve the fault tolerance of a system is the basis of Recovery Oriented Computing (ROC), which says that faults and errors “are facts to be coped with, not problems to be solved” [3]. Even though ROC focuses on MTTR, the MTTF is not ignored: Since ROC forces you to understand failures in order to recover from them, that understanding can be used to help try and reduce the number of failures, increasing MTTF.

2.3 Distributed Systems

In general, a distributed system is a collection of computers working together in order to accomplish a common goal that is larger than any single computer could accomplish in a reasonable amount of time. There are many types of distributed systems that are classified based on the types of problems they are designed to solved, as well as how they solve those problems. The systems discussed in this part fall into two categories: grid computing and stream computing.

A grid system is a distributed system that has tightly coupled components, with each computer working on a small part of the large problem. This type of distributed system is most similar to a traditional supercomputer (which has multiple processors in a single computer), since the basic problem is the same for each category of computers [7]. For example, if you had a very large array, and had many computers processing small chunks of that array, it does not matter which computer processes each part of the array, as long as the entire array is processed. In that sense, the computers in the grid are interchangeable.

A system using stream computing is different from a standard distributed system in the way it processes queries for data. With a standard query, the data source is searched once and the data retrieved is not changed; in order to get new data, the database must be queried again. The idea of stream computing is to use “continuous queries” to get data and continually update it as the data changes [2]. For example, if you wanted to keep track of your location using GPS, instead of having to send a query whenever you think you have moved, you would just use a continuous query with stream computing, and your information would update as soon as the information from the GPS changes. More conceptually, a system using stream computing gets its data from multiple continuous queries (or ”streams”) from different but related sources, and uses that data to determine a larger whole.

2.4 Remote Procedure Calls

If a system needs to perform an action that cannot reasonably be performed by one computer, that action can be broken up into many smaller steps, and those steps can be performed by different computers in a distributed system. In order to communicate between the computers, remote procedure calls (RPCs) are used to send messages between computers, and these low-level messages are represented as standard procedure calls [4]. These messages are sent from the part of the system the procedure is called from (the client) to the part of the system performing the action (the service), and the two communicate to ensure the action is performed. Even though high level programming languages such as Java disguise the use of RPCs as much as possible, the use of RPCs introduces new errors dealing with the low-level message passing, most notably when the client receives no response from the service, and these message-passing errors require additional fault tolerance.

An example where RPCs is used is online shopping. For example, if you purchase an item from a large online retailer, the retailer’s system may need to check if your credit card is still valid while also checking if the item is still in stock before serving you a web page asking you to confirm your purchase [6]. The retailer’s website, the retailer’s inventory system, the credit card company’s validation service are likely all on different computers, and RPCs are used to communicate between the systems.

The ideal case when a client sends a message is for the service to execute the command exactly once, but when the client receives no response from the service, the client does not know if the service executed the command or not [4]. In these cases, the client could simply resend the message after a predetermined timeout, in order to ensure the service executes the command at least once. However, in many distributed systems, this can cause problems. Going back to the online shopping example, having even the chance that your credit card could be charged twice while trying to buy an item is not acceptable to the customer. On the other hand, the client could attempt to check in with the service, only resending the original command after it confirms the service failed on the first command, ensuring the command is executed at most once.

3. TIME-CRITICAL GRID COMPUTING

3.1 Examples of Grid Computing Systems

Although speed is a concern for almost all systems in some sense, some systems are time-critical, either because the data set they are working with is rapidly changing or because the system is part of an inherently risky action. These systems have a strict time limit to execute the process in, and as such the goal is to maximize the benefit in that time. This benefit is measured with a benefit function [9] which varies based on the application domain. In addition to being fast, these systems need to be fault-tolerant, and that fault tolerance cannot come with a significant loss in speed.

The main example we will look at in this paper is real-time rendering of 2-D images from a 3-D data set, specifically rendering tissue volumes during surgery [9]. This is time-critical because the rendering happens in real-time. If a notable event is shown in the image (such as an abnormality in the tissue), the surgeon can request detailed information on the event. The goal is for an image to be rendered and displayed, and this has to happen in a fixed amount of time.

The secondary example in this paper is a system used for data mining, or the act of finding patterns in a large collection of data [1]. The structure of the data-mining system is that when a user starts a data-mining operation, that operation is first communicated to a global unit. That global unit then communicates that operation to a group of local units, and those local units perform the actual data mining. When the local units each finish mining their piece of the larger collection of data, that unit communicates that information back to the global unit, which integrates the data it receives from all the local units. When all the local units have completed their data mining, the global unit communicates the results back to the user.

3.2 Checkpointing and Replication

The simplest way to implement fault tolerance in distributed systems is to duplicate the entire system multiple times, so you can switch over to a backup system if the main unit fails. The easiest way to implement a duplication strategy is to run the backup systems concurrently with the main system, but that has many problems. Running many systems simultaneously will cause the performance of each individual system to fail, which particularly causes problems for time-critical systems [9]. In addition, in some systems like the data mining system, some of the actions are non-deterministic [1], which means that random actions would have to be consistent across all the backup systems, which is not possible.

In grid systems like the ones described in Section 3.1, the system consists of one large, main process that calls many small processes, and most of those small processes are called repeatedly. For example, rendering a small section of tissue is not that much work relative to the whole system. With these processes there is not a lot of state to be saved, compared with the total size of the application. Since there is not much to save, you can efficiently save that small amount of state and then propagate those changes to the backup systems in a process known as checkpointing. This checkpointing allows the backup systems to be updated without incurring the overhead of running all of the backup systems.

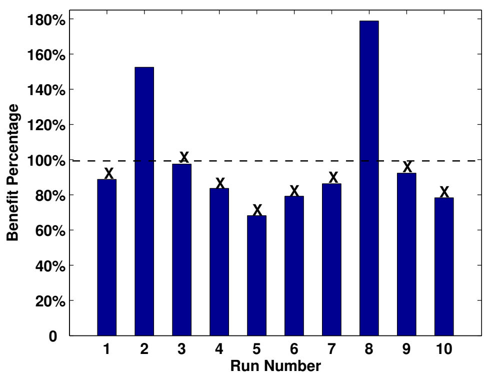
Cesario and Talia use checkpointing in a fault-tolerance strategy known as primary-backup [1], which works as described above: instead of having multiple units running, the primary-backup strategy only has one unit running at any one time (the primary unit), but there are a number of backup units ready to take over if the primary unit fails. If and when the primary unit fails during the collection of data, a new primary unit is selected from the backups. Using the primary-backup strategy, the new architecture has r global units, one of which is the initial primary unit and the rest are backup units. Each backup unit has a failure-detection unit associated with it, and these backup and failure-detection units are linked to the necessary resources during the first phase of the data-mining process.

There are three phases to Cesario and Talia's primary-backup strategy for the data mining system: checkpointing, failure detection, and recovery. During the checkpointing phase, the change of state of the primary unit is sent to each of the $r-1$ backup units, and this happens after the global unit integrates a segment of data from the local units. Failures can be detected because the primary global unit sends a heartbeat message to each of the failure-detection units periodically while processing the data, and if this message is not received within a given time, the failure-detection unit assumes that the primary unit has failed. If a failure-detection unit detects that the primary unit has failed, it awakes its associated backup unit and that global unit takes control of the data-mining operation as the new primary unit, sends a heartbeat message to the remaining failure-detection units, and continues mining from the last checkpoint sent. This primary-backup strategy protects against failures, while increasing the calculation time by only 4% compared to a system without fault tolerance.

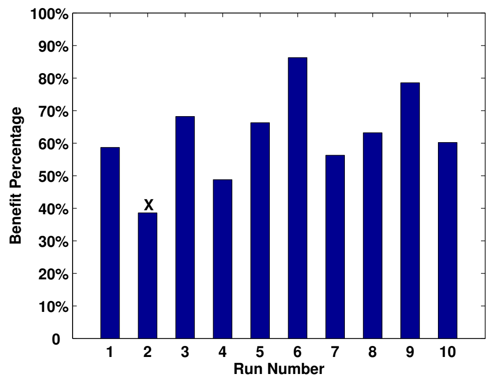
In contrast, for the time-critical systems discussed by Zhu and Agrawal, although many of the processes are small enough for a checkpointing-based strategy to work, some have too much state to be efficiently checkpointed. Instead, there are three possibilities for those larger actions. While the first possibility is to simply replicate the part of the system for that process, the other two look at the overall benefit we would lose on a failure or gain on a restart versus the time needed. If the process that failed just started, it can just be restarted, since it probably had not gained much benefit, and the time used would be minimal. On the other hand, if the process was almost finished and we can recover the results, we can just accept the benefit gained since the time taken for a restart or to execute fault-tolerance procedures would be large compared to the remaining benefit. If the process falls between these two extremes, we will use the checkpointing or replication techniques described above, depending on the size and state involved in the process.

3.3 Reliable Time-Critical Systems

The focus for fault tolerance in Zhu and Agrawal's system is obtaining reliable resources. To explain the process and algorithms for obtaining those resources, they define several concepts [9]. Each time-critical application is made of a set of services $S_1, S_2, ..., S_n$, and the application in general has a time constraint T_c. Each application has a benefit function B (as described in Section 3.1) and a baseline benefit B_0 it needs to provide in order to be considered useful. The goal is to provide the baseline B_0 within T_c, while maximizing B. For the tissue volume rendering system from Section 3.1, the benefit function is based on the error tolerance and the image size.



(a)



(b)

Figure 1: Benefit percentage of VolumeRendering system with different selection heuristics from [9]. The x represents a failed run. (a) shows the results using the efficient resources, (b) shows the results using the reliable resources. (The y-axes differ.)

Given a selection of resources Θ, there are two straightforward ways to select which resources to use while managing failures. On one hand, you can use the most efficient resources. On the other hand, you can use the most reliable resources. However, neither of these met Zhu and Agrawal's needs for maximizing benefit. Figure 1 shows the results of ten test runs for the volume rendering system under each approach, showing a percentage calculated as $B(\Theta)/B_0$; 100% or higher means the achieved benefit exceeded the required baseline benefit B_0. Using the efficient resources, two runs succeeded in providing benefit greater than the baseline benefit, but eight runs had failures and thus failed to meet the baseline benefit. Using the reliable resources, although only one run failed, none of the runs exceeded the baseline benefit, with an average benefit percentage of 70%.

When failures occur in a system, they generally do not occur purely randomly or in isolation. Each node N^i of the system has a reliability value $R_N^i \in [0,1]$, where 0 means the node always fails and 1 means the node never fails. Similarly, the connection between nodes i and j, $L^{i,j}$, has an independent reliability value $R_L^{i,j} \in [0,1]$. The possibility of failure of each component increases as uptime increases (which correlates to the MTTF), as well as when the workload on the system increases. Multiple failures can also occur, and those failures can happen over a short time period, or even simultaneously.

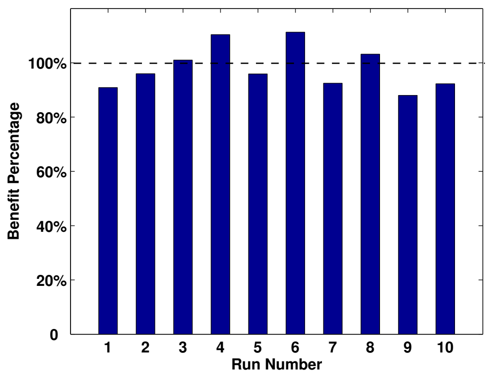


Figure 2: Benefit percentage of VolumeRendering system with multiple application copies and optimized resources from [9].

In order to maximize reliability of the system and meet the baseline benefit, the benefit function $B(\Theta)$ and the reliability function $R(\Theta, T_c)$ must both be maximized while meeting the baseline benefit and staying within the time constraint. However, since trade-offs must be made between efficiency and reliability, there is not a single solution to the problem. In our case, the solution chosen is based on a trade-off factor α, which is higher if the resources are generally more reliable[1]. The solution Θ is then the solution with the maximum weighted sum of the benefit percentage and the reliability:

$$\alpha \times (B(\Theta)/B_0) + (1-\alpha) \times R(\Theta, T_c) \qquad (1)$$

To find which resources $\hat{\Theta}$ from the full set of resources are the best for the current situation, a relatively simple algorithm is used. The algorithm starts with the objective function described as Equation 1 above, the set of services S, and the time constraint T_c. Using those parameters, the system uses an evolutionary algorithm known as Particle-swarm Optimization to determine a reasonable set of resources based on the objective function. The idea behind Particle-swarm Optimization is that there is a set of initial resource configurations, and those configurations can be represented on a coordinate axis of the two parameters (in this case, $B(\Theta)$ and $R(\Theta, T_c)$) as particles [8]. At the start of the algorithm, each of those particles moves in a random direction. Once the particles are moving, the direction they move is slightly adjusted towards the direction of its current best value, as well as the direction of the current overall best value (though the magnitude of the adjustments are random). Once the particles converge, the overall algorithm returns the corresponding resource set, which is presumably a good resource set for the given environment.

In order to test the new algorithm for obtaining resources, Zhu and Agrawal repeated the same tests as in Figure 1 using the new algorithm. In addition, Zhu and Agrawal implemented a simple fault tolerance system of replicating the system four times. The results of this test are shown in

[1]If the environment is generally reliable, more focus has to be placed on the benefits, and similarly for the opposite case.

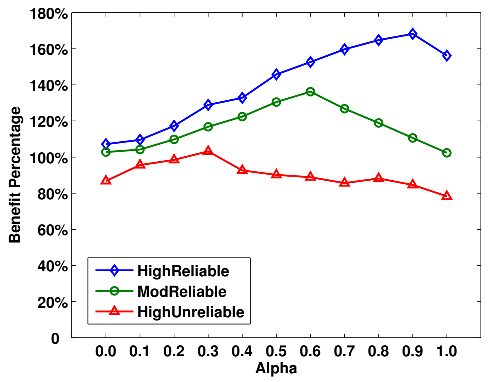


Figure 3: Benefit percentage of VolumeRendering based on α in three environments from [9].

Figure 2. Although no runs had failures, the average benefit percentage was only 96%. Since this average means the average benefit still does not meet the baseline benefit required, a more efficient fault-tolerance system was needed, and this was the checkpointing system described in Section 3.2.

3.4 Overall Fault Tolerance

In order to see the impact of choosing reliable resources, we will be looking at two specific graphs from Zhu and Agrawal. Each of the graphs looks at the performance of the volume rendering application in three environments: one where resources are generally reliable, one where resources are generally unreliable, and one in-between the two extremes. Figure 3 shows how the choice of α affect the overall benefit, including if the system meets the baseline benefit B_0. The graph shows that even though α corresponds to the environment in the best cases (higher α for reliable environments, lower α for unreliable environments), a hybrid approach is still necessary, even in the extreme cases.[2] Figure 4 shows the benefit percentage of the volume rendering in each of the three environments, based on the three failure-recovery techniques discussed: no failure recovery, only replication, and the hybrid scheme with checkpointing and replication. In all cases, the hybrid technique performs better than the pure replication technique, and, with the exception of the reliable environment, the hybrid technique performs best. In addition, the hybrid technique exceeds the baseline benefit in each case.

4. STREAM COMPUTING

As discussed in Section 2.3, a system based on stream computing gets its data from many sources, which means there are many parts of the system to manage. In addition, these parts of the system have many different divisions within themselves, and these smaller divisions of the system may or may not change the state of the larger system, and thus need to be handled differently.

An important example of a stream computing system is IBM's System S. System S is a general system which could be applied to many types of problems. The initial white

[2]The α values for 0 and 1 correspond to ignoring the efficiency and reliability of the resources respectively, and the graph shows those values get worse results.

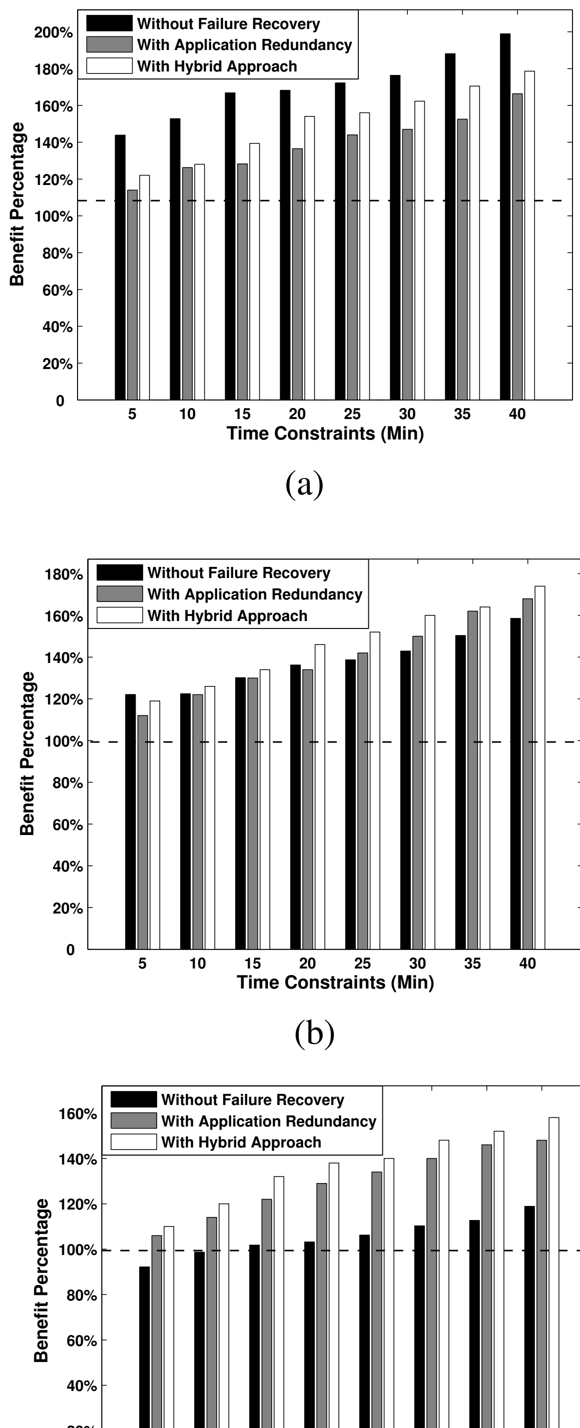


Figure 4: Benefit percentage of VolumeRendering using different fault-tolerance systems from [9]. (a) is the highly reliable environment, (b) is the moderately reliable environment, (c) is the highly unreliable environment.

paper IBM released for System S detailed many pilot programs for the system, including processing data from radio telescopes, analyzing data for financial markets, and monitoring manufactured computer chips for defects [2]. The use of stream computing by System S allows it to tackle these problems from multiple angles to get a more comprehensive solution. Looking at the financial example, to predict the direction a company's stock price could move, System S could combine the statistics from the past performance of that stock, news reports about that company, and the impact of the weather forecast on that company's business to get a prediction, then update that prediction based on how all three of those factors change.

When the user wants System S to process a set of streams to accomplish a specific task (such as predicting a stock price), this is called submitting a job [6]. Submitting a job has two major parts: first checking if the user has permission to submit the job, then actually registering the job in the larger system and processing the data. These two parts communicate with the larger system using RPC.

Since the state changes to the larger system cannot be duplicated, communicating with the larger system must use an at-most-once RPC (as discussed in Section 2.4). However, implementing a fault-tolerant at-most-once RPC is difficult to do efficiently, since in addition to worrying about registering a job more than once, you also have to worry about failures that happen during registration. The at-most-once RPC implementation in System S has two basic parts. First, the two parts of the procedure described above need to be broken into two distinct parts. Second, after the first part checks if the user has permission to submit the job, it assigns the job an ID that will be used in the second part to actually register with the larger system (like reserving a place in line), so that if the registration is interrupted, the system still knows the ID and can attempt to resume where the interruption occurred.

Most of the processes in System S were programmed in the same pattern as submitting a job. They have two parts, a part where the system is checked, and another where the system is changed. The first part needs to get an ID to reserve its place in line so the process can be completed, even if there is a failure in the second transaction. These processes do not affect one another, and once a processor completes its part of processing the stream, its failure does not affect the progress of processing the stream.

The result of this use of RPC is that the time to recover (TTR) of the system is very small compared to the total time of the operation. Wagle *et al.* showed this by inducing crashes in many possible setups of System S, and then delaying restarting the system for a set number of seconds. In these tests, the difference in the total time taken for the operations was almost equal to the artificial restart delay, showing that the TTR was on the scale of milliseconds while the total time for the operation was on the scale of seconds.

5. CONCLUSION

Through this paper, we have looked at two different types of distributed systems, each of which handles fault tolerance in a different way. The time-critical volume rendering grid system evaluated the possible resources it could use based on their reliability and efficiency, and chose which resources to use based on the needs of the given system [9]. System S used stream computing and was separated into small components to quickly restart any component that failed, while using reliable remote procedure calls to ensure changes in state of the main system were performed at most once [6]. Each of these approaches meets the specific needs of the system, while using the ideas of Recovery Oriented Computing.

The main difference in the fault tolerance of the two systems comes in the way they solve their problems. The stream computing system focuses on many separate systems that do not relate to one another, so the RPC techniques make the different parts work together. In contrast, the grid computing systems are already tightly coupled and the parts generally only interact with the main system, and not with each other (like in the data mining system described by Cesario and Talia [1]), so RPC is not worth the extra overhead. Instead, the grid computing system can use a more detailed system for selecting resources, which is not appropriate for the stream computing system since the resources are always changing. This shows that Recovery Oriented Computing can be used for many different systems, and in different ways for each system.

6. REFERENCES

[1] E. Cesario and D. Talia. A failure handling framework for distributed data mining services on the grid. In *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, pages 70 –79, February 2011.

[2] IBM. System S - Stream Computing at IBM Research. `http://download.boulder.ibm.com/ibmdl/pub/software/data/sw-library/ii/whitepaper/SystemS_2008-1001.pdf`, 2008.

[3] D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tezzlaff, J. Traupman, and N. Treuhaft. Recovery oriented computing (ROC): Motivation, definition, techniques, and case studies. Technical report, U.C. Berkley, 2002. `http://roc.cs.berkeley.edu/papers/ROC_TR02-1175.pdf`.

[4] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design Part 1*. Morgan Kaufmann Publishing. Chapter 4.

[5] J. H. Saltzer and M. F. Kaashoek. *Principles of Computer System Design Part 2*. MIT Open Courseware. Chapter 8, `http://goo.gl/hhppt`.

[6] R. Wagle, H. Andrade, K. Hildrum, C. Venkatramani, and M. Spicer. Distributed middleware reliability and fault tolerance support in System S. In *Proceedings of the 5th ACM international conference on Distributed event-based system*, DEBS '11, pages 335–346, New York, NY, USA, 2011. ACM.

[7] Wikipedia. Grid computing — Wikipedia, The Free Encyclopedia, 2012. [Online; accessed 24-October-2012].

[8] Wikipedia. Particle swarm optimization — Wikipedia, The Free Encyclopedia, 2012. [Online; accessed 25-October-2012].

[9] Q. Zhu and G. Agrawal. Supporting fault-tolerance for time-critical events in distributed environments. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 32:1–32:12, New York, NY, USA, 2009. ACM.