De-centralized Peer-to-Peer MapReduce System

CS647 Distributed Software Systems Final Paper (Spring 2009)

Omar Badran
Drexel University
3141 Chestnut Street
Philadelphia, PA 19104
Ob37@drexel.edu

Jordan Osecki
Drexel University
3141 Chestnut Street
Philadelphia, PA 19104
Jmo34@drexel.edu

William Shaya
Drexel University
3141 Chestnut Street
Philadelphia, PA 19104
Wss24@drexel.edu

1. ABSTRACT

MapReduce distributed systems typically are restricted to a subset of computers in a close geographic location. These computers are also typically pre-assigned to be either a "master" or "worker". The jobs that are accomplished are usually submitted by whoever is overseeing the operation. Therefore, the system has limited processing power by computers which cannot be easily reassigned and if there is a failure, there is no way to recover from it without human intervention.

This paper proposes a new type of MapReduce system. This system would be on a Peer-to-Peer (P2P) network and would be de-centralized. The system would employ self-configuration, setting up its topology of workers, master, and job client for each job. The system would employ self-healing by reacting if any of the three worker types fail during the middle of the process.

The result is a system which can perform MapReduce operations with access to unlimited processing power in the form of any nodes that wish to connect from anywhere. The roles of master, worker, and job client can dynamically change between jobs using the self-configuration properties, meaning that anyone who wants to submit jobs can and any node can also lead the process as master. If a failure does occur, it can use the self-healing properties to recover by replacing the failed node. If a worker fails, grab a free node or another worker and re-distribute the failed node's work to them. If the master fails, have the nodes appoint a new one, which can take over at any point during a job.

The group behind this paper produced a simulation version of this software and tested it under regular MapReduce conditions and under failures. Data is presented showing the benefits of self-healing and self-configuring, as well as discussions about taking the simulation further and applying the concepts to a real world model. There are also discussions and results presented regarding if this system is more beneficial than a typical MapReduce system.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications.

General Terms

Algorithms, Management, Performance, Design, Reliability, Experimentation, Theory.

Keywords

Distributed systems, peer-to-peer network, de-centralized, MapReduce, self-healing, self-configuration, self-*.

2. INTRODUCTION

MapReduce distributed systems are comprised of a group of servers housed at a single location. MapReduce jobs are therefore limited to running on those servers. Depending on the size of the job, the number of available servers in this location may not be enough to complete the job in a reasonable amount of time. MapReduce systems typically have one "master" and several "slave" or "worker" nodes that perform the map and reduce functionality. The "master" coordinates the MapReduce operation while the workers actually perform parts of the job.

Some of the pitfalls of this topology are that all jobs that are ran are decided within that server room, the jobs can only utilize those resources available in the area, the whole system is dependent on the single master for coordination, the job is dependent on all of the workers doing their job, it is difficult to switch the role of a node in the process even if new roles would dramatically increase the efficiency of that job, and it is not very good at reacting to failures, which will most likely need human intervention to resolve them.

Most of these pitfalls are from the technology and topology structure. For example, jobs can only be submitted from one geographical location and a limited number of servers. The jobs themselves might be performed inefficiently because it may require a much greater number of nodes than are available in this location, but the system will be forced to overwork the available ones. The whole system is dependent on the master being fully functional and dividing up jobs. If that master fails, the operation will not complete correctly or not complete at all because there is no protocol or anything to assign a new master. It is also dependent on workers doing their respective jobs. If any of the workers fail, the work must be re-assigned, further tightening the constraints of resources. Furthermore, it will most likely take human intervention to get any failed nodes working again.

This paper proposes the development of a simulated MapReduce system over a de-centralized Peer-to-Peer (P2P) network. This system will have the ability to have access to numerous servers that are connected to the Internet for both extra computational ability and job submission. Therefore, there will be many more nodes working on data than before and anyone can submit jobs for the group to work on and solve. The de-centralized approach will allow any peer node to act as the master; therefore, in the event that the current master fails, another peer node can take over. There will no longer be a dependency on any one master or any workers, as new masters can be appointed from the general node mass and new workers can be found to replace any that may have been disconnected. No workers will be overburdened, as there will be enough to split up the task into the ideal chunks necessary, choosing the ideal workers for the job. This system will employ two self-* innovations in order to further improve the MapReduce experience. The first is self-configuration, which will be responsible for choosing the master and workers and ideal chunk sizes for each job. This will make the jobs finish more quickly and more efficiently. There is also self-healing, which will be responsible for watching the workers and master and if either fail, will quickly and easily replace them, barely hindering the job that is being performed. All of these changes will make for a more open, efficient, utilized, robust, and reliable system overall, as compared to the traditional MapReduce system.

The example simulation system produced by this team had some very promising results. First, the system showed that with an unlimited number of nodes to choose from, the chunks can be split up between workers in the best way possible ("best way" is another research question altogether, but this system will not restrict whatever that number may be). With the self-* technologies that are incorporated, this not only limits the issue of node churn that comes with using a Peer-to-Peer network, but it also reduces issues of a typical MapReduce system having their master or workers fail. This paper will describe the system, compare it to the older system, describe related work, and present some empirical data from the simulation, showing the effect of failures and how the self-* algorithms limit the amount of time lost. Some tests are also run to show the main goal, which is that the de-centralized Peer-to-Peer approach, with its unlimited nodes but increased churn, is more efficient than the centralized approach with limited nodes but less churn.

3. BACKGROUND

MapReduce systems are used for very large jobs that in some way have tasks which can be performed independently of each other. A simple example of this is counting the number of words in a file or counting occurrences of a word or other traits or characteristics in a file. This is purely a parallelizable job with each portion independent because no results depend on any others. One worker can count the words in a paragraph and it will not affect the job for any other worker. Although this is a trivial job, there are much more important and complicated jobs in all fields of science which can be split up in chunks and have the results calculated in parallel. This saves a lot of time and resources. For example, if a job is one hundred percent parallelizable and there are ten workers, it can do it in about ten percent of the time as compared to one worker doing the job.

Typically, these MapReduce systems have a master, which receives the job submission from a job client. The master then decides how to split up the job and give it out to the workers. Once it is given out to the workers (the Map part), the workers do the computation on each piece and when they are done, submit it back to the master, which will re-combine all of the pieces to form an output (the Reduce part), which it sends back to the job client. This model is good, but it also has some pitfalls. For example, the master is a large single point-of-failure because it is responsible for the Map and Reduce steps. If it fails during either, everything is lost. Furthermore, by the nature of MapReduce, it is typically housed within a small geographical location, limiting the possible participants and resources.

The de-centralized Peer-to-Peer MapReduce system, as proposed by this paper's authors, aims to solve these basic problems with the "current" system and even more, as described in the "Introduction". The de-centralized approach will eliminate the single point-of-failure. The master can change if it goes down, so there is no worrying about whole process being disrupted for a long period of time, unlike with the traditional MapReduce system. Under this new system, the "master" will have much less responsibility, as the Reduce function will be moved to be a task for the job client. This is significant because a failure by the job client cripples the network much less than a failure by the master, and if the job client fails, their job submission is not as important any more. This will be described in more detail in the "Approach" section.

Through the use of the self-* algorithms that were described earlier and will be described in more detail in the "Approach" section, this new system will be better equipped to handle all points of failure and all types of failures more efficiency than the current system's architecture. The Peer-to-Peer approach, as compared to a closed network, will allow more nodes access to be participants in all roles. This abundance of extra peers, along with the more sophisticated self-* algorithms, will work to eliminate any negatives (i.e. churn) from switching to a Peer-to-Peer network and produce a more efficient system overall.

4. APPROACH

This section describes the overall simulation system produced, the architecture of the system, the self-* algorithms deployed in the system, the system's plan of implementation, and the future work and considerations section, for other researchers to use to advance the work this team has accomplished.

4.1 System Description

The proposed simulation system is a de-centralized MapReduce system which runs on a Peer-to-Peer network. The simulation system proposed here was implemented using Java. More details on the organization of the system will be provided in the second sub-section.

The goal of this project is a "proof of concept" to see if a MapReduce system can be implemented over a P2P network without a central command/control computer. It will also show novel ways to recover from inefficient/disabled nodes and show techniques for handling other factors that will occur because of the P2P network and just in general. If the system can recover efficiently from failures, this will not only eliminate any negatives of running this system on a Peer-to-Peer network, but it will yield great positives over running it on a small closed network where failures are fewer but much more catastrophic.

The system created will perform the simple task of counting the number of words in a file. One node (the job client) will submit the task, being a file that needs to have its words counted. The node currently assigned as the master will receive the job submission and start the job by assigning an appropriate number of worker nodes their tasks regarding the job and informing the worker nodes where to retrieve their data set. The workers will get the necessary input data from the node that submitted the job and store that chunk on their own system. Each worker will only take the chunk that they need, while the entire original file is kept in that submitting node's memory space. With the submitting client storing the data file rather than putting the data file in a central repository, there is no longer a chance that all jobs would be lost with one single failure. Repository replication could be implemented, but this would make the system more complex. If the submitting client fails, only data associated with that client is lost and data files from other submitting clients remain unaffected.

The simulation system will be "controlled" by a simulator object which will log events and statistics, but also control events in the simulation, such as ordering job clients to submit jobs and invoking the failure of different objects. Each node will have the ability to work as a master, job client, or worker based on its code-set. The next section will describe the system's architecture in more detail, including a run through of a typical job submission from instantiation to completion and a graphic and description of what will be "contained" in each node in the system.

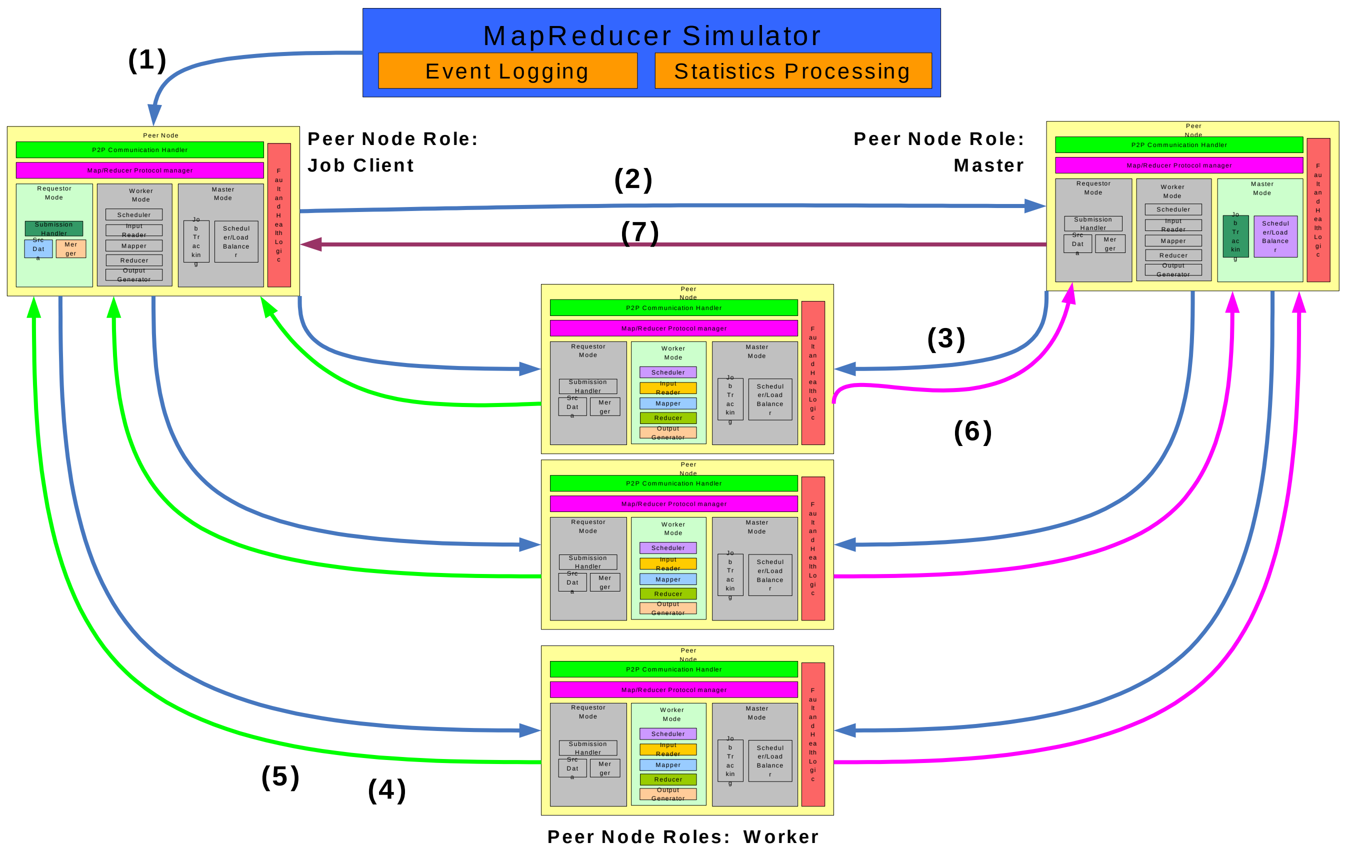
4.2 System Architecture

The Peer-to-Peer MapReduce simulator will be designed to simulate real conditions of MapReduce operations performed on a peer-to-peer network, but focusing more on the modified process and on the self-* optimizations to improve its performance. In Figure 1 below, there is a diagram which shows a run-through of a job in the de-centralized system. The following steps take place:

1. The simulator process makes a direct call into a job client node to start a MapReduce job on a certain file.
2. The job client sends a message to the current master node to tell it that it wants to submit a job.
3. The master analyzes the job, selects a number of worker nodes that should work on different chunks, and sends messages to each of these workers to tell them about their upcoming task.
4. Each worker receives the news about the chunks they are receiving to work on. Each "worker" will send a message to the job client, which will be holding on to the input file the entire time, and request their chunk. The worker, upon receiving their chunk, will begin processing it based on how it knows to complete this type of job.
5. When the MapReduce job has been completed, the worker will send the manipulated chunk back to the job client.
6. Each worker will then send a message to the master to inform them that they have completed their MapReduce task and sent the chunk back to the job client.
7. The master will send a message to the job client indicating that the MapReduce job is complete and that all of its chunks have been delivered back to it. The job client is responsible for merging the results of the job.

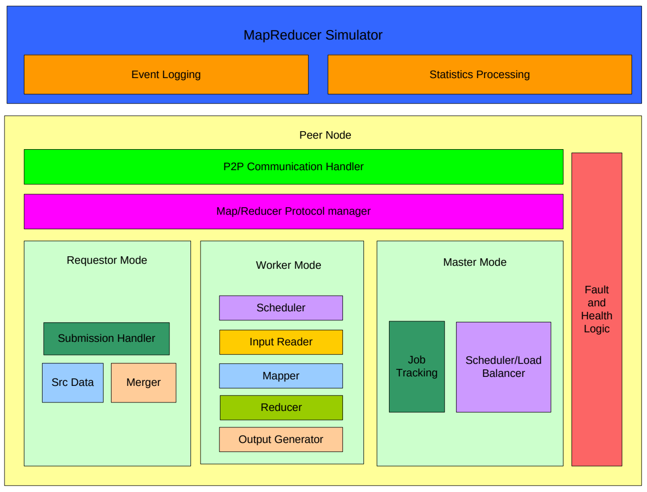
Figure 2 (on the next page) shows a graphic of the classes which will make up each node. Its parts are described in more detail:

Figure 1: Sequence of Actions for one job in P2P MapReduce Simulator



- The over-arching MapReducer Simulator, which controls event logging, statistics processing, and can cause nodes to perform actions, such as submitting a job or inducing a random failure.
- The P2P Communication Handler, which will simulate Peer-to-Peer messaging within the simulation.
- The MapReducer Protocol Manager, which will run the entire process and ensure it follows the steps, as described and shown in Figure 1.
- The Requester Mode, which is when the node acts as a job client. It contains a submission handler, the source file, and a merging method to get its output.
- The Master Mode, which is when the node acts as a master. It contains a job tracking method and a scheduler/load balancing method.
- The Worker Mode, which is when the node acts as a worker. It contains a scheduler, input reader, mapper, reducer, and output generator.
- The Fault and Health Logic, which will be used to monitor other nodes, to switch roles and to detect and react to failures.

Figure 2: Structure for all nodes in the simulator



4.3 Self-* Algorithms

The proposed simulation system will incorporate self adaptation through self healing and self configuration.

Self-healing is accomplished by monitoring the worker nodes and the master node. The master will monitor the workers by sending out a heartbeat type mechanism to all of them. The worker nodes will all monitor and respond to the heartbeat. A worker node will be declared failed by the master if it fails to respond to the heartbeat after several heart beat cycles. The master will be declared failed if a few neighboring worker nodes determine that they have not received a heartbeat from the master after a period of time, which will shift as the number of nodes in the system shifts. If a worker node fails due to loss of connectivity to the network, other fatal conditions, or from running very poorly, the failed node's computation will be re-distributed to a healthy node. If the master node fails, one of the other peers will take over as the master node and then continue the MapReduce operation from where it left off. The new master will have the global state at some time and send messages to check how up to date it is. Therefore, the overall computation can seamlessly complete despite the failures. The application framework will include a module to induce random failures throughout the simulated network in order to exercise self-healing. The self-healing algorithm will also be able to be turned off and on in order to test the effect it has on efficiency of recovering from failures.

During processing, if any worker nodes go down, the master will re-distribute their work to the other nodes. A priority scheduler is implemented to ensure that re-distributed work gets completed before any other jobs that the worker may be assigned to perform next. If the master goes down, the nodes will decide who the new master is and that node will take over the processing where it was left off. The new node may or may not be a node that currently has work of its own to do. Since the main role of the master is to assign, a new master will not delay the job majorly. Due to the design of the workers retrieving their data sets directly from the submitting client, any already scheduled MapReduce operations could be completed while a new master is negotiated. When the processing is finished, the chunks will be sent to the node who submitted where they are re-combined and the node has the solution for its job. A worker node will perform the mapping and reducing of its data set while the submitting node will merge all of the "reduce" results [7].

Self-configuration is accomplished by the peer nodes negotiating who will act as the master and the remaining nodes will be dynamically allocated as workers depending on the size of the MapReduce operation. Workers will be recruited and selected by the master based on if they are functioning and if they are needed for the job's size. For example, if the job size is 100MB, perhaps ten workers will be allocated, while a 1GB job may utilize 100 workers. The master will have a list of workers based on an algorithm using neighbors and using broadcasts. In order to evaluate the effects of self-configuration, the tasks are monitored to ensure they are completed correctly even in the midst of failures, inefficiencies, and re-configurations. The algorithm for choosing the master node can involve using efficiency values of the worker nodes. As described more in the "Future Considerations" section, the simulation uses a simple algorithm where each node will choose a random number and the node closest to zero will become master. The project's future work will include determining efficiencies of each node, allowing such a factor to be involved in this algorithm in that version.

4.4 Project Plan

This paper implemented the java application simulation solution in two phases:

The first phase implemented the de-centralized Peer-to-Peer MapReduce simulation system with a single master. Upon job submission by any node in the network, work was distributed among available peers for task completion. Self adaptation techniques were not yet incorporated in the first phase. This phase served as a baseline for the concept and to ensure the basic system is operational.

The second phase was implemented upon completion of the first phase. In this phase, the team incorporated the self-adaptation discussed in the previous sections. Figure 2 showed the system's component block diagram. Phase one was responsible for implementing the "Requestor Mode", "Worker Mode", and "Master Mode." In phase two, the "Fault and Health" logic is implemented (this includes the heartbeat, self-configuration, and self-healing) and so are the hooks allowing the simulator to control certain aspects, such as inducing failures. In this phase is also when the system was tested and evaluated, using the scenarios described in the "Evaluation" section. That section will also present the results that were found.

4.5 Future Work/Considerations

There were some basic assumptions and reductions that were made with the scope of this project only spanning three months. Some of these are outlined below, along with suggestions for others to use if they wanted to continue this research.

The main reduction made for the scope of this project was that in the simulation system, each node runs as a thread under the control of the simulator and communicates via method calls, rather than sockets. This avoids defining messages that would be sent over TCP/IP. Furthermore, the system assumed that all nodes will be connected to each other. This avoids the process of nodes forwarding messages. These simplifications are only to allow the system to overlook basic Peer-to-Peer operations. This allowed the simulation system to focus on the Self-* solutions and handling scenarios, not on message routing. The next version of this simulation should do away with some of these assumptions.

Another reduction was that the system only focused on detecting nodes that fail or disconnect. This will be done using a heartbeat pinging mechanism to detect a failed worker and the absence of a ping and collaboration by the workers to detect a failed master. The "Fault and Health" monitor located within each peer node module will be responsible for the heart beat mechanism. In a future design, overall node efficiency will be calculated. This is valuable because node work can be distributed differently based on nodes that have not failed or disconnected, but are simply performing poorly. Furthermore, when the master fails, the efficiency of nodes can be taken into account when appointing a new master. In the simulation for this paper, the new master decision has a place where efficiency values can used in the calculation, but right now it uses purely random numbers to "represent" efficiency.

5. EVALUATION

The system was evaluated by using the simulator to deploy and record different scenarios and their results. The simulator employs statistics and event logging. The logged information depicts the state of the system when faults are introduced. For example, it will show when the master has failed and which peer node has taken over. The statistics will show the duration of MapReduce operations and how they are affected by various conditions.

Several scenarios were developed and used to test the system. There are scenarios to test the proposed MapReduce system without any of the Self-* functionality implemented and scenarios that were run to test the two Self-* applications and their implementation and what effect they had on the overall performance with and without failures.

The success of the system will be judged by how it can perform different scenarios, all of which will rigorously test the features of the system. It will pre-dominantly test the features which make this system novel and significant as compared to the systems described in the "Related Work" section.

The test data used for the MapReduce operation was approximately 6.5 kilobytes in size. The system was configured to have either three or six worker nodes.

5.1 Testing Basic M/R Implementation

The first set of tests are run without any Self-* functionality implemented. The reason for this is to obtain a baseline set of results so that the impact of implementing Self-* functionality can be evaluated.

Furthermore, these tests will compare a closed network MapReduce to a Peer-to-Peer MapReduce. Failures cannot be tested here, but it is safe to say that the latter system reacts much better to them. For example, if the master were to fail, it would cripple a closed network job. If workers were to fail, it also would take much longer to re-distribute the work. This is the advantage of the de-centralization and self-* algorithms.

One thing that was tested is the limitless amount of nodes available in this team's system versus the limited amount in typical MapReduce systems. The team tested our simulation with a limited number of nodes and a job that required the use of all of them, to simulate the closed network. To simulate the P2P network, there are more than enough nodes for the job and some failures of workers thrown in to represent churn. The results are displayed in the table below:

Table 1: Times for Different Number of Workers on the Simulation System with no Self-* Algorithms (Baseline)

Simulation Type	Time
3 worker node system	12.410 Seconds (Avg)
6 worker node system	13.312 Seconds (Avg)

5.2 Testing the Addition of Self-* Algorithms

The second set of tests simply test the benefits of the two self-* algorithms, by running a successful scenario and scenarios with different nodes failing and the self-* algorithms correcting the problems. The variables that can change are self-healing and self-configuration processes running if a master, job client, or worker fails. The team tested the simulation against workers and masters failing, as if the job client fails, the job would simply be aborted, as no nodes want the results of the operation anymore. The simulation results are presented in the table below. More detailed results and a graph are presented in the Appendix.

Table 2: Time Comparison between Scenarios with no failures, with a Master failure, and with a Worker Failure

Scenario	Time
No Failures (3 workers)	13.975 Seconds (Avg)
No Failures (6 workers)	22.106 Seconds (Avg)
One Worker Failure (3 workers)	25.870 Seconds (Avg)
One Master Failure (3 workers)	15.628 Seconds (Avg)

5.3 Discussion of Results

As Table 1 and Table 2 shows, the times follow the wrong trend when adding more nodes to the system. The team has attributed this to the slowdown caused in the simulator by the passing of additional messages and also the fact that true parallelism isn't achieved. The team feels that this is the cause and that this problem would be better alleviated in a real world P2P network. Therefore, the results of the tests were inconclusive to determine if the decentralization and P2P network approaches, with the extra nodes but also extra churn, were beneficial changes. The team, however, does suspect that such a scenario will play out much differently if further testing on both a better equipped simulation and in the real world were conducted.

As Table 2 shows, while the times for the failures are more than the "No Failures" scenario, they are not a lot longer and are much better than the "typical" MapReduce scenario, in which if a master or worker failed, the system would either not recover at all without human intervention or any self-* algorithms implemented would take just as long, if not longer, because it has fewer nodes to select from to help in work re-assignment or master elections. In a closed network, there is a much greater probability that the worker that would need to take on extra work or become master is already doing another job, while in the P2P solution there is a large probability that an "idle" node could be used, making the self-* solutions along with this system's architecture ideal.

The de-centralized, P2P aspect of this paper's system enhanced the robustness and scalability of the system. There were more workers (and potential masters) available and jobs could be submitted by anyone. This, the author's believe, out-weighs any drawbacks for choosing a P2P network, such as non-locality of data with respect to the workers and the churn of peers/workers which may disappear at any time. One goal of this system was to show through evaluation that the number of workers will outweigh the churn. While our results in Table 1 did not conclusively show this, the team believes that this will be confirmed if further testing was done. The issue of nodes disappearing or disconnecting is the same if the system was not P2P, but just will occur much more frequently. The self-* algorithms used in this system address this concern by accounting for all failures. Table 2 shows the system does a decent job of addressing failures. Therefore, this new system, with its de-centralization, P2P network use, and self-* algorithms, is the perfect combination to improve upon typical MapReduce systems.

6. RELATED WORK

There are several notable MapReduce and related systems that exist, such as Hadoop, SETI@home, and Skynet.

Hadoop is a Java framework used to implement MapReduce and is currently used in Yahoo web searches. These systems are based on a network of computers connected via a local network rather than a P2P network. It also is a centralized system.

SETI@home incorporates a centralized master node which distributes chunks of work to a P2P network of workers. As a result, worker nodes can only be workers, so there is no option for recovery if the master fails. Furthermore, jobs can only be submitted from the central system.

Skynet is a de-centralized, open source Ruby implementation of Google's MapReduce framework. It is adaptive, fault tolerant, and has only worker nodes. If the master fails, it has mechanisms to reassign its responsibilities, but only certain other nodes can take over for it. As a non-P2P system, tasks that fail on a particular node can only be reassigned within a defined set of workers. There are no outside job submissions or processing available.

The system that this paper describes has advantages over both SETI@home and Skynet. SETI@home's centralized topology leads to an inoperable system should the master fail. This paper's solution of a de-centralized system will maximize usage and availability such that any node may submit jobs, be a worker, and be the master. By incorporating the P2P architecture in the design, the paper's system is superior to the Skynet system, since the workers are not limited to a pre-defined set.

In [4], a similar topology to this paper's is described. The authors discuss master failure and recovery as well as worker failure and recovery. One key difference between this paper's system and what is described in [4] is that this system will only have one master. This approach will allow more available workers and still maintain the same level of functionality. Some of the fault tolerance ideas from [4] may be used, but this system employs more de-centralization by having one master, which any node can become if necessary.

7. CONCLUSIONS

MapReduce systems in the traditional sense have many pitfalls, such as a centralized master, limited resources, and inability to easily recover from failures. The de-centralized Peer-to-Peer MapReduce simulation system with self-configuration and self-healing produced by this paper's team takes large leaps towards addressing all of these issues.

Theoretically, all of the problems of a typical MapReduce system are solved by this new architecture. There are no single points of failure, the data and control (master) are de-centralized, there are limitless resources to choose from to run job submissions with the ideal number of nodes, and failures are quickly addressed.

The simulation system created and the tests and their results in the "Evaluation" section show that there are significant time gains from the self-* algorithms in place, under different failures. It also shows that this new system performs better with limitless nodes to choose from.

The work can be further extended by improving upon the team's simulation system, implementing more components of it to remove assumptions it currently makes. These assumptions are listed in the "Future Considerations" section. Other than improving the simulation, actual real-world Peer-to-Peer network systems and more testing and evaluation could be used to validate the results of this paper.

8. ACKNOWLEDGMENTS

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

The team would also like to thank Professor Valetto for helping to guide the project in the right direction and raise issues for the team to consider.

Finally, the team would like to thank Yiannis Nicolaou and Dale Seybold Jr. for providing valuable insight and excellent questions during our peer review session with them.

9. BIBLIOGRAPHY

[1] Cardona, K., Secretan, J., Georgiopoulos, M., and Anagnostopoulos, G. 2007. A Grid based system for data mining using MapReduce. http://cygnus.fit.edu/amalthea/pubs/Cardona_Secretan_TR-2007-02_AMALTHEA.pdf

[2] Dean, J., and Ghemawat, S. MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM, Vol. 51, No. 1, January 2008*, pp. 107-113. http://www.scribd.com/doc/240523/MapReduce-Simplified-Data-Processing-on-Large-Clusters

[3] Hadoop, http://hadoop.apache.org/core/

[4] Marozzo, F.,Talia, D., and Trunfio, P. Adapting MapReduce for Dynamic Environments Using a Peer-to-Peer Model. Extended Abstract. http://www.cca08.org/papers/Poster7-Domenico-Talia.pdf

[5] SETI@Home, http://setiathome.berkeley.edu

[6] Skynet, http://rubyforge.org/projects/skynet

[7] Yang, H., Dasdan, A., Hsiao, R., and Parker, D. Map-Reduce-Merge: Simplified Relational Data Processing on Large-Scale Clusters. *SIGMOD'07, June 12-14, 2007, pp. 1029-1040.*

10. APPENDIX

Extra Data for Evaluation Sections 5.1 and 5.2:

Table 3: Ten Program Runs, "No Self-*", 3 Workers

Run	Submit Time	Done Time	Dif (milli-sec)	Diff (sec)
1	5079	17484	12405	12.41
2	5078	17400	12322	12.32
3	5074	17385	12311	12.31
4	5063	17512	12449	12.45
5	5062	17458	12396	12.40
6	5062	17629	12576	12.57
7	5062	17392	12330	12.33
8	5047	17484	12437	12.44
9	5047	17488	12441	12.44
10	5047	17484	12437	12.44
		Avg	**12409.5**	**12.41**

Table 4: Ten Program Runs, "No Self-*", 6 Workers

Run	Submit Time	Done Time	Dif (milli-sec)	Diff (sec)
1	5079	18450	13371	13.37
2	5074	18267	13193	13.19
3	5062	18360	13298	13.30
4	5062	18390	13328	13.33
5	5047	18375	13328	13.33
6	5078	18375	13297	13.30
7	5063	18375	13312	13.31
8	5062	18375	13313	13.31
9	5047	18354	13307	13.31
10	5047	18423	13376	13.38
		Avg	**13312.3**	**13.312**

Table 5: Ten Program Runs, "No Failures", 3 Workers

Run	Submit Time	Done Time	Dif (milli-sec)	Diff (sec)
1	5063	18829	13766	13.77
2	5062	19000	13938	13.94
3	5078	19234	14156	14.16
4	5062	19266	14204	14.20
5	5062	19218	14156	14.16
6	5125	18797	13672	13.67
7	5062	18968	13906	13.91
8	5078	19218	14140	14.14
9	5062	18968	13906	13.91
10	5063	18969	13906	13.91
		Avg	**13975**	**13.975**

Table 6: Ten Program Runs, "Worker Failure", 3 Workers

Run	Submit Time	Done Time	Dif (milli-sec)	Diff (sec)
1	5063	29828	24765	24.77
2	5062	29734	24672	24.67
3	5062	33796	28734	28.73
4	5063	32250	27187	27.19
5	5063	28547	23484	23.48
6	5078	28594	23516	23.52
7	5062	29828	24766	24.77
8	5047	28672	23625	23.63
9	5063	35125	30062	30.06
10	5062	32953	27891	27.89
		Avg	**25870.2**	**25.870**

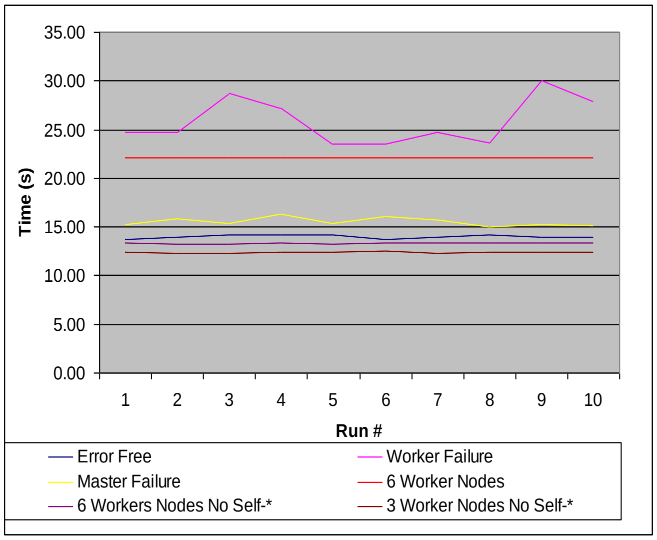
Table 7: Ten Program Runs, "Master Failure", 3 Workers

Run	Submit Time	Done Time	Dif (milli-sec)	Diff (sec)
1	5063	20282	15219	15.22
2	5063	20891	15828	15.83
3	5062	20406	15344	15.34
4	5062	21437	16375	16.38
5	5078	20453	15375	15.38
6	5063	21188	16125	16.13
7	5062	20781	15719	15.72
8	5062	20031	14969	14.97
9	5063	20282	15219	15.22
10	5062	20156	15094	15.09
		Avg	**15628.2**	**15.628**

Table 8: Ten Program Runs, “No Failures”, 6 Workers

Run	Submit Time	Done Time	Dif (milli-sec)	Diff (sec)
1	5062	27187	22125	22.13
2	5047	27141	22094	22.09
3	5063	27157	22094	22.09
4	5062	27187	22125	22.13
5	5062	27125	22063	22.06
6	5063	27156	22093	22.09
7	5062	27187	22125	22.13
8	5063	27172	22109	22.11
9	5031	27156	22125	22.13
10	5063	27172	22109	22.11
		Avg	**22106.2**	**22.106**

Figure 3: Graph Showing how the Scenarios in Tables 3-8 Compare to Each Other over their Ten Runs



De-centralized Peer-to-Peer MapReduce System

Omar Badran, William Shaya, and Jordan Osecki

The following is the document describing precisely how this project's work reflected the original proposal and factors that have turned out differently in any way:

The project for the most part followed our proposal perfectly. The team stayed with the basic ideas, used the same research throughout, and found ourselves only clarifying certain things that were left purposely undefined in the proposal, pending further research. Examples of this include some of the algorithms that were going to be deployed at each step of the MapReduce process.

The simulation itself stayed relatively the same to our proposed model. The basic ideas stayed the same. The node structure with all of the classes and methods outlined did not change throughout our implementation. The self-* algorithms and methods of self-healing and self-configuration chosen also stayed the same.

The largest change that occurred between our initial proposal and our finished product was that the team decided to follow the approach of a MapReduceMerge-type system, since this would allow the master to only have to assign jobs, the workers to only have to do their jobs, and the submitter node to only have to receive and merge the individual results. The team saw this as an opportunity to gain even more advantages over the traditional MapReduce system. By using this new approach, it significantly reduces the work of the master, making the system more balanced and less dependent on the master. With this less work, there is no longer any reason for the master to handle any of the data, so the job client submitting the job is responsible for holding on to it. This is beneficial because if a central repository or the master held on to the job and either failed, it would be catastrophic. But in this scheme, if the job client fails, it is losing the job that it owns, so it has incentives to stay up and if it loses its data, then the rest of the workers and master do not care because they had no stake in the submitted job.

Another change that occurred between the proposal and final implementation was that the team set one of its main goals to be to prove that the churn of a P2P network is more than offset by the available number of nodes. It was always a goal to focus on how the self-* algorithms, de-centralization, and Peer-to-Peer network would be benefits over the traditional MapReduce closed networks, but it became apparent for the team early that Peer-to-Peer would bring some negatives along as well as the positives, so it would be vital to show that the positives outweighed both the negatives of it and negatives of the traditional system.

The final change between proposal and final implementation was that the team settled on algorithms for master assignment and worker assignment, which involved efficiency values. In the proposal, the team discussed efficiency values as future work. However, for the final implementation of the system, the team chose to incorporate random values in its algorithms which would represent efficiency values. In the future, these values will be some measure of a node's efficiency to make the network even more efficient.