Using a Failure History Service for Reliable Grid Node Information

Catalin Leordeanu and Valentin Cristea
Faculty of Automatic Control and Computers,
University Politehnica
of Bucharest
{catalin.leordeanu,valentin.cristea}@cs.pub.ro

Thomas Ropars, Yvon Jégou
and Christine Morin
INRIA Rennes-Bretagne Atlantique,
Rennes, France
{tropars,yvon.jegou}@irisa.fr,
christine.morin@inria.fr

Abstract

The need for reliability in Grid Systems is a difficult challenge which is very important in the context of highly dynamic systems composed of thousands of nodes. Failure management is a key component in the attempt to provide such a reliable environment. This approach is based on the existence of accurate failure information about the nodes in the Grid which is very difficult in large scale systems. This paper proposes a failure history service used to share failure information which is critical to the management of resources in large scale distributed systems, thus improving the overall reliability. This novel service ensures that the information about the current state of a node, as well as its failure history, is as accurate as possible even when facing a large number of node failures. This solution aims to increase the reliability of Grid systems by providing accurate data which can be used to analyze failures over time.

1 Introduction

Failure management is an essential component in distributed systems, used to provide reliability to the environment and to running applications. Due to the increase in size and complexity of distributed systems there are a lot of challenges that need to be studied.

An important part of failure management is the actual detection of failures which refers to the action of identifying a loss of functionality of a resource in the distributed system, usually a running process. Hardware and software instability, unreliable networks can be the cause of such events.

In this paper we describe the main elements necessary for failure management, the most used theoretical solutions and propose a novel service used to share failure information. This work focuses on Grid Environments and was designed as a module for the Vigne[10] middleware.This system provides the Grid nodes with the ability to receive reliable information about the past failures in the Grid System, as well as to determine the current status of a node. There are a number of challenges to the design of a such a storage system, mostly related to scalability and accuracy. In a Grid System we can have large numbers of nodes so any centralized system would be inefficient. Therefore, we took advantage of the Vigne architecture which provides useful mechanisms for node management.

Failure detection is also challenging in itself due to the characteristics of large scale distributed systems and also the difficulties in establishing reliable communication channels, while maintaining a low overhead. The existing failure detectors must not add any considerable delay to the normal operation of the distributed system so the low overhead is an important concern in the design of the detectors.

Another cause for concern is the existence of different active failure detectors for the same nodes in the Grid. These failure detectors may employ different mechanisms to detect node failures so they might also provide different feedback about the same suspected node. The correlation of information from different failure detectors to create a clear picture about the status of the existing nodes in the Grid is also a focus of our research.

This approach is very useful for Vigne, especially during the process of resource selection, when an Application Manager must choose the most reliable nodes to run an application. Determining which are the most reliable nodes is only possible if we can gather accurate information about the history and current status of the available nodes. This is the purpose of our solution, as we will describe in the following sections. Another contribution of this paper is a novel method to determine the most likely state of a node based on the available information from the failure detectors.

Also we provide different test cases, as well as experimental results which validate our solution. The tests were carried out on large numbers of nodes, in a very dynamic

simulated Grid System, with large numbers of node failures and restarts.

The rest of the paper is structured as follows. Section 2 presents the context of our research. It describes in general terms the challenges of failure detection in large scale distributed systems, as well as other projects related to our work. Section 3 describes Vigne, its features regarding failure management and the approach used to integrate our solution. The proposed solution is described in section 4 and the experimental results are presented in section 5. Section 6 concludes this paper and outlines the areas for future research.

2 Context and motivation

This paper directly addresses a very important subject in large scale distributed systems. It aims to improve reliability through the management of failure information. This subject has been the focus of many research projects with the purpose of improving reliability.To better understand the problem we first need to look at the existing failure detection mechanisms which form the basis of our solution.

2.1 Failure Detection Mechanisms

Failures can range from simple crashes and network failures to byzantine behaviours where the affected node starts exhibiting arbitrary behavior. It is however impossible to detect the difference between the crash of an application and that of the network connections. They are both seen from the outside as a complete loss of communication with the running application and any external monitor cannot tell the difference without the use of an alternate route to that node in the distributed system. There is also a possibility that an entire node of a distributed system will fail, or only the application which is being monitored, which then fails to respond to messages.

Therefore there are a number of possible causes and contexts for failures. There may also be different failure detectors which would only detect certain types of failures or use certain detection mechanisms. This leads to the problem of receiving different information from different failure detectors.

There are a number concepts that need to be considered when evaluating failure detection systems. Detectors can be classified according to the following characteristics:

- *Completeness.* Describes the ability to detect failures. The easiest completeness property to achieve is weak completeness is when a failed process is suspected by every correct process. [3]
- *Accuracy.* Is the ability of not mistakenly detecting a failure when a process is still functioning correctly.
- *Speed.* This characteristic represents the time between the actual time when the failure occurs and the moment the failure detector suspects the process. This usually depends on the timeout values of the messages sent by the failure detectors.
- *Scalability.* Represents the ability of the failure detection system to maintain its performance with the increase of the number of nodes within the distributed system and the number of messages which are sent.

Most approaches are based on unreliable failure detectors that continually monitor the targeted resources and report suspected failures. A resource or a node in the distributed system is usually suspected as having failed when it doesn't send a periodic "I'm alive!" message or doesn't respond to a query until a timeout is reached. This is an unreliable mechanism due to the fact that messages over a network can have large and often unpredictable delays.

Regarding the performance of failure detection systems the overhead which is added by the additional messages is also an important factor. For this reason the UDP protocol is used instead of TCP because of the speed and the use of a smaller number of messages than it would take to have a permanent connection between the applications. This reduces the overhead but unfortunately UDP messages can have large variations in latency and can be lost so it makes the failure detection system even more unreliable.

The main goal of failure detection is to provide information about unforeseen events and also aid in the survivability of running applications. In this case failure detection helps to identify these abnormal behaviours of the processes so an appropriate action can be taken. Possible actions as a result of the detection of a failure are usually application dependent. Most often we encounter one of the following actions:

- Stop the entire application
- Ignore the failure
- Allocate new resources and restart the application
- Use replication and reliable group communication primitives to continue execution

Each of these actions has its advantages and disadvantages. Most of them depend however on other existing mechanisms or on the type of the applications running in the distributed system. For example if an application only requires that a single component returns a correct solution to a problem ignoring a failure is an efficient way of dealing with unforeseen events.

Restarting an application is usually a good way to keep the application running and increase the probability that it will finish successfully. The disadvantage in this case is

that all of the computations until the failure are lost and the application will start from the beginning. If a checkpointing mechanism is also present then the restart will be from the last valid checkpoint so the lost computation time will be minimal.

Other advanced recovery methods could also be employed[4]. Besides attempting to restart the application a recovery system could also use multiple copies of the application on different resources.

The failure detection system needs to function correctly for any type of application running on the distributed system. There are however types of applications which may have their own failure detection mechanisms or which could provide additional information for the failure detection system. For example a failure detection system could use the information gathered from MPI applications to efficiently detect failures.

2.2 Related work

Failure detection has been the focus of many projects[1]. Challenges in this field usually orginate from the unreliability of the failure detectors.

Since our work is focused on to the management of multiple active failure detectors and also the storage and retrieval of the information received from them there are few projects with similar goals. In the work presented in [2] the authors propose a scalable monitoring system which shares some of our goals. It also presents the importance of having reliable monitoring information to be able to take important decisions about the large scale distributed systems.

In [7] the authors propose a coordinated infrastructure for fault-tolerant systems through which different components can share fault information with each other. Their approach has the goal of detecting complex faults but it does not store the fault information, nor does it draw any reliability conclusions based on the failure patterns of different components. It also presents a novel way for various software systems to plug into their infrastructure and share information. This solution is efficient but it is oriented more to the small scale high end computing domain, which is not too well suited to the use in large scale distributed systems.

Another interesting approach is the one presented in [6]. It outlines the main concerns about process migration in the presence of faults, which is also one of the applications for our work, using the stored failure detection information.

There are also approaches related only to the storage architecture. The article introducing PAST[5] presents a distributed reliable storage system built on top of Pastry and it also uses the neighbors of a node to store information. It is also considered a reliable system but it is used only as a general storage, without the use of the failure detectors which may be active for the nodes. An similar approach based on PAST is also used for caching in [12]. The system proposed in this document is closer to [12] since the management of failure detection information could be considered similar to a caching system. These approaches are not focused on the improvement of reliability so they don't bring any failure management mechanisms as in our approach.

3 Vigne

Vigne, which is described in detail in [10], is a middleware, whose goal is to ease the use of computing resources in a Grid for executing distributed applications. Vigne is made up of a set of operating system services based on a peer-to-peer infrastructure. This infrastructure currently implements a structured overlay network inspired from Pastry[11] and also provides Single System Image, Self-healing services and Self-Organization.

As we will describe in the following sections, we used the Pastry overlay to build our Failure History Service which was designed as a module to be integrated in Vigne. Pastry has a very simple mechanism to assign nodeIDs which we use to define the storage distribution of the failure information over the nodes in the Grid.

Another important thing about Vigne is its approach to application execution. When a user sends an application to be run on the Grid Vigne creates an Application Manager which will then decide which nodes of the Grid will run the application and will monitor its execution. An application manager acts on behalf of the user to run efficiently the application and to ensure that it terminates correctly, despite node removals and failures. This is the step where accurate failure data can really influence the performance and the reliability of the system. If the Application Manager has access to the failure history of each node and it can be sure that the information is accurate then it can safely choose the best nodes for the application. In this case it can choose the nodes which have the lowest probability of failing while that application is running.

In conclusion, Vigne is a complex project, capable of offering high levels of fault tolerance, as shown also by the experimental results. The implemented mechanisms are also scalable, capable of handling highly dynamic configurations with large numbers of nodes.

4 A Failure History Service for Vigne

In this section we will discuss the details of the proposed failure management solution for the Vigne architecture. It involves the design of a distributed sharing system for failure detection information for the Vigne middleware, or Failure History Service(FHS). The first part of this section describes the theoretical aspects of the proposed solu-

tions, followed by implementation details and testing scenarios within the Vigne middleware.

This subject is very important for resource selection, determining the current state of a node and other crucial operations in large scale distributed systems. Our goal is to provide a service which will gather and combine the existing failure detection mechanisms and provide reliable information about what is happening in the distributed system. This solution is built on top of Vigne.

This solution makes use of existing failure detection mechanisms, gathering failure information from all the available detectors. From the point of view of other nodes this service is seen as a simple storage element, which can be queried to receive accurate information about the history of a node or its current state.

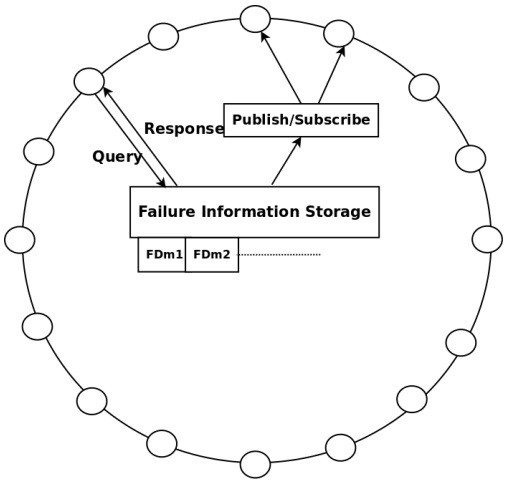

Figure 1. Capabilities of the Storage System

The functionality of this service can be divided in two parts as shown in figure 1. A node will be able to query the Storage System and receive the complete failure history of any other node or just the reliability index. The failure history contains is a list of all the past failures which may have affected that node. The algorithm used to determine the reliability of a node is described in the following sections. It will also be possible to receive notifications about the changes in the state of a node using the Publish/Subscribe mechanism.

Since we are relying on data from various failure detection mechanisms they could provide conflicting data so the information on the possible state of a node may not be entirely accurate. In this situation an application could subscribe to receive notifications about the state of another node when it is available with a certain degree of accuracy.

Through this system reliable information about the state of a node will be available. It is also configurable by design so it is useful for applications which require accurate information about nodes as well as applications which do not need near-perfect information and wish to save bandwidth by sending less messages, thus making sure that each application runs according to its needs.

4.1 Storage Architecture

This architecture provides a distributed storage environment for failure detection information. If each node would store its own failure history the information would be vulnerable to failures and it would also not be available when that node is unavailable. To solve these problems, the failure information and history about a node will be stored on the k nodes with nodeIDs closest to the source node. The value of k is global and fixed for the entire distributed system. This approach ensures that the data from the failure detectors will not be lost if any node crashes or experiences data loss.

Using this scheme, any application will be able to request information about the node it's interested in by sending a query to any other available node. If the targeted node does not have the requested information in its local storage then the query will be forwarded to another available node in the vicinity of the target, as seen in figure 2. The storage system also functions correctly even if the target node has failed and is unavailable at the time of the query because the response will be given by one of its neighbors which also contains all the failure information of the target node.

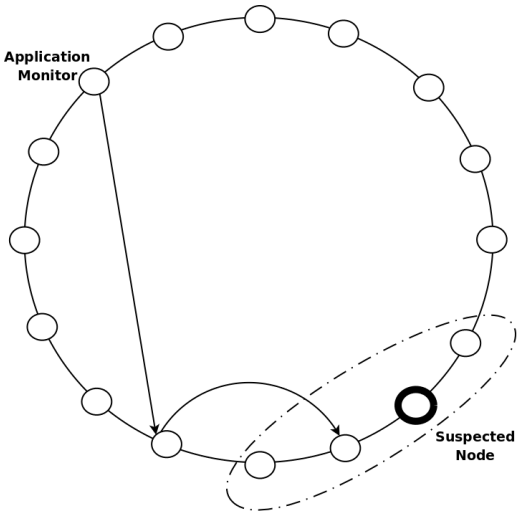

Figure 2. Storage distribution

An application could subscribe to receive notifications about the state of another node when it is available with a certain degree of accuracy. As we know, no failure detector can offer information that a node has definitely crashed. It can only provide a list of nodes that are suspected to have crashed with a certain degree of accuracy. Due to the fact

that different failure detectors can reach different conclusions based on the behavior of a node in the distributed system, the failure history of that node must also store the accuracy with which the failure information was received based on the type and number of failure detection mechanisms which were used.

There is a difficulty in ensuring the consistency of this history as a node can have a different nodeID when it recovers from a failure. In this case we can also use the hardware address of a machine to identify it, along with its nodeID. If a node appears in the distributed system with the nodeID of a failed node and has the same hardware address then it is considered that it has recovered from failure and the storage system will append its failure history accordingly.

The functionality of the storage elements on each node could be divided into the following parts:

- *Receive a notification from a failure detector.* The node will then decide how to add this new information in the history. If the identity of the node is known, it will consider that the status of the node has changed and append the history for that node. As a timestamp for this notification we will use the local time at the arrival of the notification.
- *Receive information about a node entering the system.* This event appears when a new node enters the system and receives a new nodeID. It can be the same node that previously had the nodeID which has recovered from a failure or it can be a new node. If the hardware address of the target node is different from the one already in the history then it will consider that it is a different node with the same nodeID and refresh its failure history.
- *Store the new information in the node history.* The system will compute the accuracy of the new information with the methods provided in this section and append it to the history of the node.
- *Notify interested users about the status change.* Using the Publish/Subscribe system the new information about the node will be published which will reach the applications interested in the status of that node with the desired degree of accuracy.
- *Send update messages to neighbors.* Since the *k* closest nodes will need to have the same history information of the failed node we need an update mechanism to ensure the consistency of the storage system. Each node will request updates from a neighbor when it returns from a failure.

There is also the question of keeping the history information for nodes that may not return to the system. It would be a waste of storage space to keep indefinitely all the failure information for nodes which have failed for a long time so a timeout mechanism was also designed. This timeout can be in the range of several days and triggers the deletion of that node's history, unless otherwise specified by a system administrator.

An example of the failure history for a given node is shown in figure 3. The different available failure detectors may send conflicting information and the storage system must determine the probability that a node has failed and which information to store in the failure history for each node. Since between *T1* and *T2* only a small part of the available failure detectors suspect that the node has failed that failure probability is low. This probability is higher betweeh *T3* and *T4* when all the failure detectors agree that the node has failed.

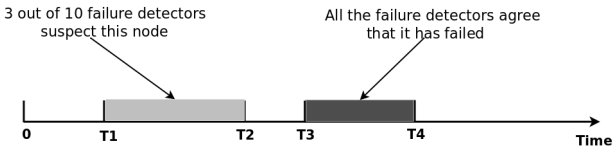

Figure 3. Failure history of a node

To achieve this goal we considered an accuracy index for each failure detector. Each failure detector receives this accuracy index based on its type and performance which is between 0 and 100 where 0 represents the most inaccurate failure detector and 100 the most accurate. Currently this property is assigned manually but we will develop an adaptive algorithm to evaluate the accuracy of each failure detector based on its responses in relation to the other detectors in the system.

We can determine the probability *P* that a node has failed by combining the inputs from different failure detectors. For example we consider a system with *n* available failure detection mechanisms with their accuracy values D_1 to D_n. We can consider the responses of the failure detectors as a vector *R*, where *R[i]* is 1 if the failure detector *i* suspects that node of having failed and 0 otherwise. The probability that a node has failed can be determined using the simple formula:

$$P = \frac{\sum_{i=1}^{n} D_i * R[i]}{\sum_{i=1}^{n} D_i}$$

This formula determines how likely it is for a node to have failed, based on the available feedback from the active failure detectors.

The information obtained using the mechanisms described in this section are also useful for resource selection. For example, if an Application Manager wants to distribute the components of its application on various nodes it needs

to choose the one which is most appropriate. One of the factors for this choice is reliability, the probability that the node will not fail during the executions of that application. One way to determine this is through the analysis of the failure history of the node. A very simple way to do this would be to determine the percentage of time that the node has been functioning since the beginning of its history information. Other, more complex, methods could also be explored.

4.2 Update Mechanism

We chose a performance driven mechanism to maintain the consistency of the replicas. The purpose of this mechanism is to guard against data loss which we assume will only happen if it is associated with a failure event which is detected by an active detector.

In this context the update mechanism is executed by the node recovering from a failure to make sure that it updates its history with the events which took place while it was down and also to recover any lost data which may have happened during the failure.

After the failed node recovers and receives a nodeID it will simply choose the closest neighbor and sends a request for the entire history of the nodes it should store. This request has the same format as the one from a client which was described earlier and the target node cannot determine any difference between the two cases. After it receives the reply it will then merge the event lists from the message with its existing data. Only after this process is finished the node will start responding to queries, to make sure that it will not send any responses with incomplete data.

This simple approach was implemented due to the small number of messages involved, which leads to a small overhead and also involves a small number of active nodes. It is also very easy to integrate since it only uses existing client queries. Choosing the closest node can be done randomly from the list of neighbors or a recovering node may also send multiple queries to different neighbors and store only the first reply.

From the test cases presented in the next section we determined that this mechanism provides a good degree of fault tolerance and is sufficient to restore the event history of nodes in the event of a failure, unless there are $2k$ simultaneous failures of neighboring nodes.

4.3 Implementation details

In this section we discuss the details about the modules which were implemented, as well as the testing methods used. The system was implemented as a simulation in SPLAY[9], which is a system for deployment and development of distributed applications. It allows the deployment of splay daemons which simulate the functionality of nodes in a distributed system. The daemons can also run, stop and monitor the existing applications. A central controller is also used which manages a database containing the information about the existing hosts and applications.

Applications are developed using the LUA language[8]. The language is very easy to use and also allows for rapid development of distributed applications. It can also be extended with additional functionality with the use of C functions. All of this is aided by a set of libraries which provide the functionality for common distributed systems applications, while exhibiting a low overhead.

In order to present the implementation of this simulation we need to describe the interfaces used by the modules of the system. Note that each node is identified using its nodeID and hardware address. For the storage elements the following actions are possible:

- *Remove.* Completely deletes the history of the node identified by nodeID and hardware address.
- *Add.* Adds new information to the history of that node. In order to update its history you must specify the nodeID, hardware address, type of the new information(failure or join) and source (identity of the failure detector). If the storage system does not contain any other information about that node a new failure history is created.
- *GetState.* For a requested node it returns the identity of each available failure detector and whether that detector suspects the node of having failed or not. In this simulation since there is no global time it will actually return the latest stored events which are close enough that they can be considered to be related.
- *GetHistory.* Returns the complete failure history of the node. Just as represented in figure 3 this history is a list of time periods, each with a different probability that that node has failed depending on which failure detection mechanism suspects it.

From the outside of a node, only the *GetState* and *GetHistory* methods will be available. Also, a node will be able to subscribe to a notification about a change in the status of a node, through a simple *Publish/Subscribe* system. Using this system an application manager could ask to receive a message when it is known that a certain node has failed with a required degree of probability. The formula used to determine the probability that a node has failed is described in the previous section.

This approach for the Publish/Subscribe mechanism is very useful because if an application requires very accurate failure detection information, at the cost of speed, it will subscribe with a higher accuracy and so a notification will be sent only when a large percentage of the failure detectors

have concluded that the target node may have failed. On the opposite side, it may not be as important to have very accurate information but an applicatio may need to be notified as soon as possible so a notification will be sent even when the information isn't very accurate.

Since this is a simulation we also implemented a mechanism to randomly generate failure detection information from various detectors. It determines the probability that a certain number of nodes will fail at a given time and also each failed node is given a 50% probability that it will return in the Pastry ring after a failure for each step of the event generation process. A dedicated SPLAY node is used for this task and it periodicaly generates and broadcasts events regarding various node failures, as well as the events from new nodes joining the system. It also contains all the information about the existing failure detectors and their accuracy and sends the events on their behalf.

Each event contains the following information:

- *Timestamp.* This is the local timestamp from the node that receives the event from the failure detector and is used to order the events.
- *Failute detector ID.* The identity of the failure detector which sent the notification about the event. This field is irrelevant if the event is of a new node entering the system since no failure detectors are involved in this case.
- *Type.* The type of event. This can be either a failure of a node or an event signaling a node that joins the system.
- *NodeID.* The identity of the node which is the cause of the event. In this case the NodeID is the same one which is used in the Pastry DHT.
- *Hardware address.* The hardware address of the node. Since a node in Vigne may fail and then return with a different NodeID we use the hardware address as a second method to identify the node.

If there are multiple events from different failure detectors which refer to the same failure then they will be correlated according to their timestamp. Since the timestamp associated with the event is only a local one and the notifications may reach different nodes with different delays the events are considered to be correlated if they are closer than a small predefined time interval t. This means that if two or more events need to be stored and they the distance between them is smaller than t it will be considered that they are different reports regarding the same event.

5 Experimental Results

The tests which we deployed focused mainly on proving the scalability, accuracy and performance of the proposed system. The first tests that we ran for this simulation were to determine the exact state of the nodes at any given time. We can see in figure 4 how the number of stored events on a two different nodes increases as the active detectors send more feedback about the suspected failures, as well as the events concerning node recovery.

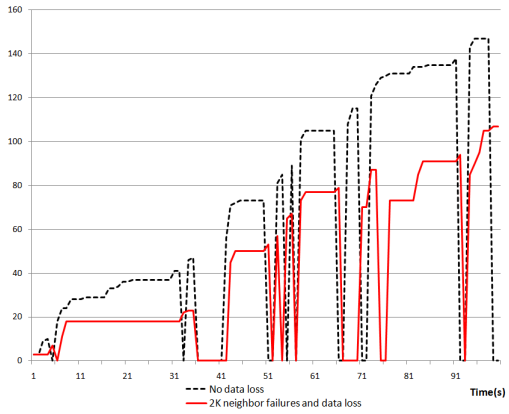

Figure 4. Number of stored events

These tests were performed with a set of 50 nodes, under moderate churn with each stored event containing the data described in this paper: the type of event, the ID of the failure detector that sent it, the nodeID of the node which is suspected, local time when the event was received.

We can see that the total size of the stored data is not very large, but it does present a steady increase. This would require an increasing amount of space as the storage system would run for a longer amount of time. A policy for the archiving of old or irrelevant information could solve this problem but it would not be critical due to the availability of storage space. Also, from this experiment we can see how the tested node failed and lost all its data during the time it was monitored. During the simulation each failure was associated with a temporary loss of the stored data, which was recovered using the update mechanism after the node recovered from the failure and rejoined the Pastry overlay.

We also conducted experiments where more than k nodes failed on each side of the monitored node in the overlay. As expected, since all the replicas of the failure history were lost, it was impossible to recover them so when the monitored node restarted and went through the update process it was only able to recover the list of events from its neighbors, not its own history. These scenarios are present in figure 4 where after a failure the node recovers with only a part of the history it had before since its own history is lost after

the failures of all the available replicas. This proves that the proposed mechanisms are fault tolerant for less than *2K* simultaneous failures.

The next experiments are relevant for the overhead generated by the storage and filtering operations. We measured the total CPU time used by this storage module, in the same conditions as the previous experiments, as we can see from figure 5.

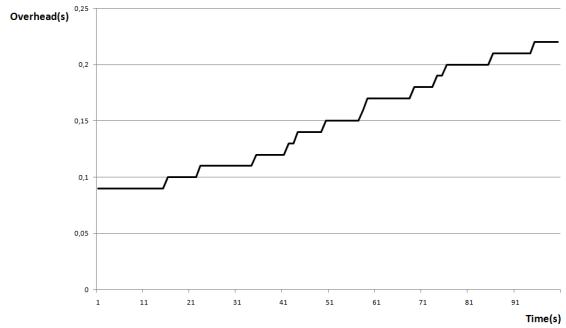

Figure 5. Overhead measurements

The overhead added by the storage and filtering operations is very small and would not interfere with the normal operation of a middleware such as Vigne. The value of our solution comes from the the advantages it brings, of increased reliability and accurate information about the nodes in the Grid System and also the fact that it achieves this goal without any major decrease in overall performance.

6 Conclusions and Future Work

This paper presents a general overview of failure detection mechanisms and also proposes a solution for a failure history service which shares the failure information between the nodes in a Grid system. This is a novel approach to node history management and represents the main contribution of this paper, along with novel methods to determine the current state of a node based on the information received from active failure detectors. The proposed solution is described as part of the Vigne project.

We also proved that our solution is scalable and can efficiently receive a large number of events. It can also aggregate the information received from all the available failure detectors so that the stored information is accurate.

As future work we intend to develop an adaptive algorithm to adjust the accuracy of each failure detector based on its responses in relation to the other failure detectors in the system. Another possible research direction is the evaluation of the reliability of a node based on its failure history. We could also use statistical methods to determine the probability that a node will fail in the immediate future.

We also intend to make our solution resilient against various attacks. Since the failure information which is stored is critical to application deployment we need to ensure that it is stored in a secure way and also that a malicious node cannot insert false data undetected.

Acknowledgments

The research presented in this paper is supported by national project DEPSYS Models and Techniques for ensuring reliability, safety, availability and security of Large Scale Distributes Systems, Project CNCSIS-IDEI ID: 1710.

References

[1] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, pages 99–125, 2000.

[2] L. Baduel and S. Matsuoka. A decentralized, scalable, and autonomous grid monitoring system. *Principles of Distributed Computing*, pages 1–15, 2008.

[3] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, pages 225–267, 1996.

[4] C. Dabrowski. Reliability in grid computing systems. *Concurrency and Computation: Practice and Experience*, 2009.

[5] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. *Hot Topics in Operating Systems, Workshop on*, 0:0075, 2001.

[6] C. Engelmann, G. R. Vallee, T. Naughton, and S. L. Scott. Proactive fault tolerance using preemptive migration. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:252–257, 2009.

[7] R. Gupta, P. Beckman, H. Park, E. Lusk, P. Hargrove, A. Geist, D. K. Panda, A. Lumsdaine, and J. Dongarra. Cifts: A coordinated infrastructure for fault-tolerant systems. *International Conference on Parallel Processing (ICPP)*, 2009.

[8] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho. Lua—an extensible extension language. *Softw. Pract. Exper.*, 26(6):635–652, 1996.

[9] L. Leonini, E. Rivière, and P. Felber. Splay: distributed systems evaluation made simple (or how to turn ideas into live systems in a breeze). In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 185–198, Berkeley, CA, USA, 2009. USENIX Association.

[10] L. Rilling. Vigne: Towards a self-healing grid operating system. *Proceedings of Euro-Par 2006*, August 2006.

[11] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, pages 329–350, 2001.

[12] A. Rowstron and P. Druschel. Storage management and caching in past, a large-scale, persistent peer-to-peer storage utility. *ACM SIGOPS Operating Systems Review*, 35:188–201, 2001.