Towards a Self Healing Information System for Digital Libraries

Vamshi Ambati, Raj Reddy

(*Institute for Software Research International, Carnegie Mellon University, 5000 Forbes Avenue, PA-15213*)

E-mail: vamshi@cmu.edu, rr@cmu.edu

An important area of focus in complex systems development has been the capability to adapt to variable runtime environmental resources and to accommodate runtime system failures. The research in this area is broadly termed as "Self Healing" and has gained increasing attention in the recent past when dealing with complex systems. Digital libraries have gained popularity for the rich features that they provide when compared to traditional libraries. As more users begin to make use of digital libraries, addressing downtimes of these valuable resources becomes a high priority. Operating 24/7 and providing access to digital content to anyone, anytime from anywhere in the world, results in a continued rise of administrative overhead for system monitoring and needs continuous human intervention. Given the volume of information and the huge infrastructure of modern libraries, continuous manual system administration is quite costly and is not feasible too. In this paper we propose a self healing digital library system as the solution to this problem. We present the approach of adding self healing capabilities to an existing digital library project, the Digital Library of India. We also propose a self healing framework that enables a successful reuse of our approach to other architecturally similar Digital Library Systems.

Key words: Digital Library, Self Healing Systems, Failover, Configuration Management

1. INTRODUCTION

Digital Libraries have received wide attention in the recent years allowing access to digital information from anywhere across the world. They have become widely accepted and even preferred information sources in areas of education, science and others [1]. The rapid growth of Internet and the increasing interest in development of digital library related technologies and collections [4][2] helped accelerate the digitization of printed documents in the past few years.

To keep pace with the flood of new information, the digital libraries will require the means to collect, digitize and make it available digitally. We have sustainable storage technologies to address the capture of this digital information, and web technologies to host the data for the usage of all. However, as more data comes in, handling the resources becomes unmanageable. Also in the long run due to changing environments, faults in software or continued usage of the hardware, a system yields to failure conditions. Especially when digital libraries grow in volume, the infrastructure and hardware resources of deployment grow in proportion, leading to troubles in manual maintenance. Recovery of systems, from daily and common failure conditions like network outages, or simple wear away of hardware would involve lot of human attention and effort. Though the data is present in the storage media and ready to be made use of, the service is unavailable for users for longer periods of time, which is unacceptable as the primary goal of being accessible and useful for end-users is now lost. Hence, though assembling the data and making it available for easy access are identified as the most important phases [3], high availability nevertheless becomes a compelling quality attribute for digital library systems. Maximizing the availability and reliability of the system and services once deployed, by automatically diagnosing, isolating, and recovering from faults is the need of the day.

An important area of focus in the complex systems development has been the capability to adapt to variable runtime environmental resources and to accommodate runtime system errors. This area is broadly termed as self healing systems. It has gained increasing attention in recent past when dealing with autonomic computing [7] systems. The term "Self Healing" introduces a new interesting and promising research field. It simplifies the task of composing, configuring, and deploying high-availability solutions and continuously monitoring their availability. However this area is not well explored in depth, not yet delineated and final targets are not well identified. Researchers have different and contradictory opinions and ideas about autonomic computing on the whole and self healing in particular.

In this paper we discuss self healing and the relevance to digital libraries. We do not intend to disambiguate the existing terminology or goals of the

field as such. We attempt to provide a simplistic approach for providing a digital library system with self healing capabilities for addressing the problem of maximum availability of the system for maximum benefit of the users. We discuss this in the context of the Digital Library of India (DLI) project, a large scale book digitization project with a goal of digitizing and hosting online a collection of one million books by 2008. The project focuses on providing access to digital content to anyone, anytime from anywhere in the world in an efficient and user-friendly manner. Based on the experience from development and deployment of large scale digital libraries, we propose "Self Healing" as a technique to ensure quality of service in the phase of faults. We also discuss in detail the proposed strategy and framework for self healing of systems in general and digital libraries in particular.

The rest of the paper is organized as follows. In section 2 we talk about the Digital Library of India project and its architecture that we base our self healing approach on. Section 3 describes the proposed approach for adding self healing capabilities to a system. Section 4 discusses a strategy for self healing in digital library systems and also proposes a framework for reusing the technology for other digital library systems. Section 5 concludes with some future work intended in this direction.

2. DIGITAL LIBRARY OF INDIA

In this section we first give a brief introduction to the project in context, the Digital Library of India Project (DLI). We also describe the status quo of the architecture of the project which we extend with self healing capabilities in the next sections.

Overview

The Digital Library of India project was initiated in the year 2002, with motivations from the Universal Digital Library project[8]. With a vision of digitizing a million books by 2008, the Digital Library of India (DLI) project aims to digitally preserve all the significant literary, artistic and scientific works of people and make it freely available to anyone, anytime, from any corner of the world, for education, research and also for appreciation by our future generations. The project currently digitizes and preserves books, though one of the future avenues is to preserve existing digital media of different formats like video, audio etc. The scanning operations and preservation of digital data takes place at different centers across India called Regional Mega Scanning Centers (RMSC). The RMSCs themselves function as individual organizations with scanning units established at several locations in the region. Responsibilities of a RMSC include regulating the processes of procuring or collecting the books, distributing across scanning locations maintained by it, accumulating the digitized content from the contractors operating at those locations and hosting the same. Hence the DLI project is a congregation of RMSCs, operating parallel and independently at distributed regions across India.

Architecture

Each Mega centre hosts the books that are scanned in the locations maintained by it. Currently there are three operational mega centers. The architecture adapted by each RMSC is similar to the one shown in figure 1. The digital objects are preserved on Terabyte servers which are clustered as a data farm. Each server in the data cluster hosts all the digital objects preserved on it, through an Apache web server. The cluster is powered by Linux and enhanced by LTSP, an add-on package for Linux that supports diskless network booting. This option of diskless network booting helps us boot a server without having to devote any space for storing the system specific and operating system files. This set up is economical and also easy to manage, in a way that we can add or replace data nodes in the cluster instantaneously without the need for operating system installations and configurations. We have customized the kernel in LTSP to support hard disk recognition and USB hot plug, and to run a light weight Apache web server.

As shown in the figure 1 the 'Linux Loader' machine runs a copy of this distribution of the Linux with LTSP. Each data server in the data cluster downloads the kernel over the private intranet and boots from it. Data can be copied onto a server in the cluster over the network or through USB interface. The servers implement a hardware based RAID to contain disk failures which adds to the reliability of the system. The 'metadata server' is a repository of the complete metadata of the books which is in XML. XML has been chosen for its important role in interoperability. Wrappers present on the metadata server automatically populate the database from the xml metadata. Along with the metadata of the book, the database also contains pointers to the location of the book in the data cluster.

The portal has a front end using which a user can login and query on the metadata to retrieve books he wishes to read online. A caching mechanism deployed on the metadata server helps us cache similar

queries posed to the database and return the results promptly. When a user requests to view the complete book content, the location of the book in the data cluster is gathered from the database and content is retrieved over http requests, from the particular server in the cluster and is broadcast to the user. The 'proxy servers layer' between the Data cluster and the portal also has a caching mechanism enabled that handles repeated requests to the book pages and ensures quick response times.

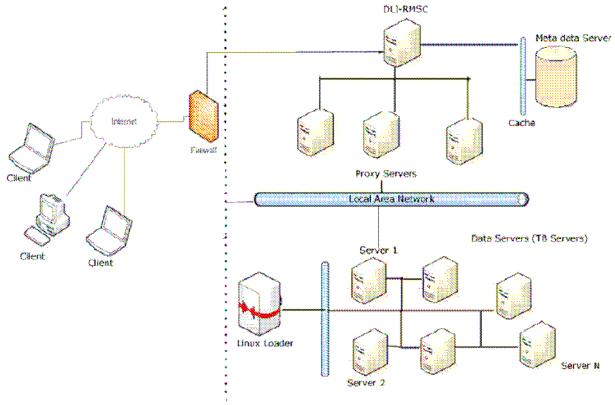


Figure 1 Digital Library Architecture

3. AN APPROACH TO BUILDING SELF HEALING SYSTEMS

The self-healing objective is to minimize all outages and to keep the applications up and available at all times. This strategy includes maximizing the reliability and availability in each hardware and software product to maintain continuous availability of applications and systems. For a system to heal itself, it must be able to recover from a failing component by first detecting the failed part, taking it off-line, fixing or isolating the failed component, and reintroducing the fixed or replacement component into service without any application disruption. A few systems also deal with on-line repair of the failed component without having to disturb the running system.

Requirements Analysis

Understanding the functional and non-functional requirements is the preliminary phase before applying self healing capabilities to a system. We first need to understand the system behaviour, so that we can understand whether the system is working according to the specification or behaving out of the specification.

Functional Requirements: An important technique that has proved to be useful in understanding and prioritizing functional requirements is by prioritizing the stakeholders and then picking the most important and most required functionality for self healing. For example in DLI, the end user is the most prioritized stakeholder of the system, and so the functional requirement of 'reading books online' naturally becomes the top priority on the self healing agenda. There are other functions like 'data synchronization' and 'administration' which are associated with other stakeholders of the system, and since they were of low priority and need not require the extra overhead of making the process self healing too. Also the failures of such functions do not affect our prioritized stakeholder, the end user. Most faults do not manifest, the ones that manifest, do not have affect on the stake holders we have in mind.

Also we may have to elicit a new set of requirements, to sketch the strategies for self healing. Also not all the functionality can be made self healing, hence we need to analyse the requirements that can be dealt with in the system.

Non-Functional Requirements: Revisiting the non-functional requirements of the system is important too. For example we may have to work in the limitations of the system like the turn around time for failover can not be less than the timeout of TCP requests, as the monitoring that helps us identify the faults might depend on TCP request and timeouts. Also, the strategy of self healing may introduce certain overhead on the system, which could possibly affect other quality attributes of the system like performance etc. So looking into these tradeoffs between the highly available quality attribute and the performance attribute is necessary. In DLI, the concern was less on performance, but more on availability of the resources to be accessible around the clock. Finally, the term highly available is vague; we need to be clear as to how much availability is expected and the down time of the system that is tolerable. This helps us plan on the healing strategies accordingly.

Probably the last but not the least, we need to elicit the kind of self healing strategy that best suits the system. If the stakeholder requires that the system focus on performance, then the self healing strategy should concentrate on constantly monitoring the environment and system for performance properties and heal accordingly based on load balancing or the like. However, systems that focus on security have to deal with different strategies altogether. Hence under-

standing of such requirements is quite essential before deciding upon the approach.

Fault Model

Building a fault model for the system is the next task towards self healing. The Fault Model can be broadly seen as a compilation of the faults that are targeted for and realizing a system boundary of acceptable behavior. For such a task the knowledge of the behavior of the system is critical. Experts should understand the difference between faults and features, faults that can be resolved and faults which can not, faults that can transient (which do not persist over a long time) and faults that are persistent. Such knowledge is only obtained by close observation of the system over a period of time. Only when the expert can clearly realize the functionality of the system, he would be capable of judging its faulty behavior and also categorize faults accordingly.

In the Digital Library Project, the following is a fault model identified and which motivated us towards a proposal for a self healing architecture.

Content Cluster problems: The content cluster consists of low commodity servers, which boot up over the network. As shown in figure1 the content cluster contains the 'Linux Loader' which servers as the machine that helps the booting up of the rest of the servers in the cluster. Most of the problems in the project have been related to the maintenance of this cluster. The NFS server that is run on the Loader machine could malfunction under severe load or congestion problems. The DHCP server has a similar threat, and the failure of either of them prevents the cluster machines from delivering content.

Third party Software Problems: Most of the software used in the Project is third party software like the Apache server, JBOSS, MySQL among others. Though the threat of failure of these third party components is lower it is not to be neglected. Also there is always the possibility of these malfunctioning due to changes or variations in the environment they run in or the server they run on.

Hardware Problems: Disks containing the data are perhaps the most sacred elements in the Digital Library Project. However normal hard disks degrade over time and are prone to faulty behavior over the long run. Though maintaining a backup for each data disk is a strategy, we can not rely on the same for complete recovery as it's a time consuming process and at the same time not manageable given the large number of disks and the frequent rate of access of data. Similarly the servers that host the disks are prone to degradation and fault behavior too. Though the server failures are quite uncommon with probably one server failing in a year; with a hundred servers, the chance of one failure per day is quite high and should be addressed.

Environmental Changes: Along with the above mentioned problems, software running in an environment is incumbent upon the environmental changes. Network outages inside the cluster and power outages are frequent among others.

The above discussed fault model of the system does not subsume the faults caused due to bad software building process or programmatic bugs and errors. These have to be taken due care in the building of the system of concern. Self healing capabilities discussed here, are built on top of the system and concentrate more on the faults of the resources used by the system and the third party software or the environment.

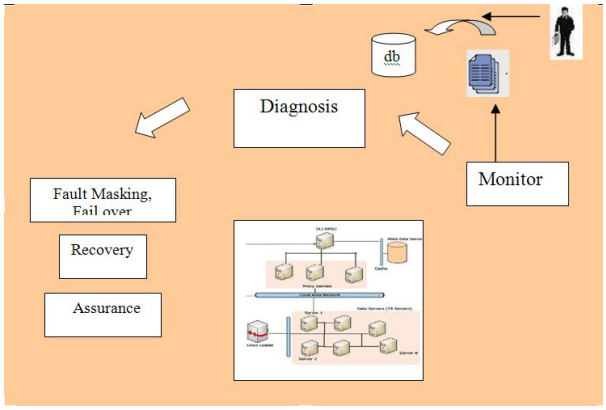
The Self Healing Strategy

Most self healing implementations approaches perform some or all of the following – monitoring, interpretation, analysis, diagnosis, failover, recovery, and assurance. The paradigm of healing could be internal where components are equipped with mechanisms of healing themselves or an externalized approach where monitors are present that are responsible for healing of the system. Externalized approaches are the default best when dealing with legacy systems as they do not entail any modification of the component. Our strategy is mostly based on externalized healing with a few components also equipped with internalized healing. For example, the servers implement a RAID setup which can be seen as a self healing mechanism for the hardware. However it is not a fool proof solution and our externalized approach with redundancy takes care of the faults that occur beyond the hardware RAID. In the following sections we describe in brief the self healing strategy that we use in the DLI project and also represent it as a framework that can be used by other similar systems.

4. SELF HEALING FRAMEWORK FOR DLI

We now describe how we make use of the fault model and the understanding of the requirements and deploy a self healing mechanism suitable for the Digital Li-

brary project. The requirements and the constraints drive our self healing mechanism. Our framework is supported by component redundancy. We move on with effective probing mechanism, fault diagnosis, fault masking and fail over.



In the DLI, the redundant components are the database, the content servers, the DHCP, the NFS servers. The following will still be there whatever you pick.

Monitoring

The primary step towards self healing is to detect the faulty behavior in the system. A system can be continuously monitored and deviation of the faults can be detected. However the monitor is not aware of or is not responsible for detecting or identifying anomalous behavior. The job of a monitor is to deploy probes to detect and report about interesting parameters and the runtime behavior of the applications in the system. Monitoring a system usually is done constantly in a background process. A probe could be a command or a transaction like a ping or traceroute command, an email message, or a web-page request. Probes are usually sent from an external machine called the 'monitor' to a server, client or any network element in order to test the availability of service of the element. The probe technology is dependent upon two techniques: bounded retries and periodic announcement broadcasting. These probes could be used to return a set of results which can then be used to calculate the parameters of concern, like latency or bandwidth etc. It could also not return a significant output other than a simple "OK" message to find out the availability of the element.

In systems like the DLI, along with manually deployed probes, we could also use the requests from end users to alert and update us on the availability of the system resources. If a end user requests for a service, and is unable to get a response, this could be used as a probe to trigger the diagnosis of the situation.

Diagnosis

Diagnosis has been addressed with very simple strategies, for example based on fixed mappings from misbehavior to adaptations methods. Slightly more sophisticated strategies select adaptation procedures by iteratively applying possible solution tactics until the problems are solved .Currently our diagnosis is centralized upon the fault semantics database. Once we have identified the faults that could occur in the system, we populate the database with the failures conditions under which machines operate in a faulty manner, and the possible tactics of fault masking, alternate component addresses and scripts that need to be run when the fault conditions match the populated conditions. This machine understandable database is the "Fault semantics database". Maintain a fault semantics Database which also consists of assurance mechanisms. For each fault, we have a scenario which tests the correction of that particular failure. Store the Information in form of a language or Grammar to decide what problems occur and how.

Diagnosis is probably the phase that requires more attention in future and that can lead to substantial improvements in self-healing systems. The ability of precisely locating the source of misbehavior can greatly improve the identification of suitable adaptation strategies. The results of the automated diagnosis are used to initiate self-healing activities such as administrator messaging, isolation or deactivation of faulty components, and guided repair.

Fault Masking and Failover

Once a fault occurs, masking the fault is necessary for any system and Content Delivery systems in particular. Fault Masking is the essential for a transparent recovery mechanism. It helps a system keep the user unaware of the fault that has occurred, while the system repairs it. One of the masking strategies is to have component redundancy as much as possible and fail over to a different component for the service. Component Redundancy is a concept introduced for addressing the self healing requirements of a system. Any resource, if needed to provide undeterred and continuous service needs a redundant copy of the resource somewhere in the system. It means the presence, at runtime, of multiple components or possibly variants providing identical or equivalent services. Only one of the redundant components providing a service is assigned, for that service. The selected variant is referred to as the active component

variant. Redundant components should have provisions to be added, updated or removed at runtime. Hence, as long as at least one worker does not continuously fail, all jobs will be completed.

The important components in the DLI project that need redundancy are the Database that runs at the backend and holds the metadata information of the books and the cluster that serves the actual content of the books. As mentioned earlier we run MySql database for holding the metadata and so we could easily have a replicated database using the replication feature provided by MySql. The content cluster has other components like the DHCP server and the Linux Loader which need redundant components in the system. Also the DLI project is hosted at three other locations in order to address issues of network outages and irrecoverable problems.

One issue to be taken care of while implementing a fault masking and fail over technique is that the presence of "state" of any form in the client request or the server system. In the DLI project, the servers are completely stateless. They do not require any kind of intermediate state or parameter values, to process the request of a client. However the client request contains some amount of state, which is the preferences of the end-user or the identity of the book viewed. If a fault occurs on a server trying to cater to the request of the client, then the 'failover server' (which is usually one of the three locations that hosts DLI content) has to take care of serving the client with those same parameters. This is achievable if we maintain some state on the server and transfer it to the 'failover server', which is complex. A simple solution is passing a pointer to the failover server and indicating the client to retry the request. Hence, we need to understand that complete transparent failover is a desired feature, but the feasibility of such a failover is largely dependant upon the type of failure that occurs.

Recovery

After a system has detected the faulty behavior, diagnosed and identified the fault and potentially masked it by degradation or fail over, we must now recover the failed component to its full operation. We can infer from this that the number of simultaneous faults that a system can handle depends upon the level of redundancy we use and the time of recovery from a failure. Recovery strategy also varies widely in complexity depending upon the "state" held by the component/resource. For a component with state, recovering from a failure involves issues such as integrating newly committed components into ongoing processes, "transferring" resources by transferring system state into them, or taking action to bring the system to a clean known state before proceeding with operations.

Most of the failures in a web based system can be recovered by a reboot. Also results of several studies [5][6] and experience in the field suggest that many failures can be successfully recovered by rebooting, even when the failure's root cause is unknown. Not surprisingly, today's state of the art in achieving high availability for Internet clusters involves circumventing a failed node through failover, rebooting the failed node, and subsequently reintegrating the recovered node into the cluster. Hence for a set of the failure conditions, the recovery strategy is a remote-reboot, and re-initialization of the server.

The methods of recovery of the failed components in the DLI project involve one or more of the following – restarting service, rebooting server, or requesting manual help. The current area of focus in the project is 'detection' of the failure occurrence so that action could be taken promptly. If the component is a data disk or a server and it is a hardware failure, a mail is sent to the administrator who then promptly replaces the failed component and does the needed. Each server hosting data is configured in RAID5 and we also maintain a redundant copy of the complete data on external storage media, for data restoration in the event of irrecoverable crashes. If the failure is due to a third party software service component like the NFS or the DHCP or the Apache then it requires a service restart or a system restart in order to recover. In our setup the one component that has state attached to it is the 'Linux Loader' server which runs the NFS and the DHCP servers. Since, a redundant component of the same runs in the system at any given point, before a recovery of this instance care needs to be taken that the internal configuration or state is synchronous with the current active running component.

Assurance

Every system requires assurances of some level of functional and non-functional correctness for normal operation. Self-healing systems additionally require a way to ensure that such functionality is maintained after recovery or fault occurrences.

In the DLI project, the assurance mechanism is more concentrated on the functional requirements and the correct operation by the specified system behavior. Hence scripts that check for this functionality are run immediately after the recovery phase. The information of the assurance scripts that need to be run are also embedded in the fault semantics database. A notable aspect here is the overlap of tasks

between the “Monitoring” phase and the “Assurance” phase. Most often we would like to assure for things that broke during the monitoring phase.

Human Interaction

Self healing is intended to shift the responsibility of dealing with failures from people to technologies. However, though an autonomous self healing system is desirable, it may not be feasible as of the current day, due to systems being built in a component based mode, where each component is sometimes obtained from a third party and the system has less knowledge of the component. Hence we try to involve the human wherever needed and not possible for system to make decisions.

In particular, in our model, human involvement comes in the form of a data-driven feedback loop between monitored logs and the fault semantics database. This ensures a fool proof fault semantic database, as this is the central element of the fault diagnosis and the self healing strategy of the system. Also if a system fails to pass the assurance test after the recovery, the human is called for, to correct the situation and also modify the conditions and recovery actions in the database.

5. CONCLUSION

We proposed a simplistic approach to add self healing capabilities to an existing system and discussed the application of the process to the Digital Library of India (DLI) project. We also explained the failure model and the architecture of the project and the constraints that support this problem. We also explain in detail the phases of a self healing system that makes efficient use of the process to implement the self healing strategy for any Digital Library system.

6. FUTURE WORK

The area of self healing has gained popularity in the research community and several approaches have been proposed. Frameworks like the IBM Autonomic computing initiative [7] have been released. We plan to experiment on how easily such frameworks can be applied to a digital library system with few constrained requirements. Also our approach currently addresses self healing for high availability, but very soon we would like to look into issues of healing during security breaches and threats to data and issues of low performance. The fault semantics database is currently populated by a human and may not include all the conditions and scenarios for the faults occurring. We need to provide good intuitive interfaces that make it as easy as possible to capture expert knowledge about rules, constraints and assurance policies. Machine learning techniques will have to be considered and incorporated for improving self healing techniques over time. Finally, we would like to work towards developing a full version of the framework proposed which could be reused by different other digital libraries which would need to add self healing capabilities.

ACKNOWLEDGEMENTS

The first author would like to acknowledge, the discussions he had with Mr.Vasu Balla of Oracle, India and Mr. Edward Walter, the Systems Manager of the ULIB project that helped in surfacing the fault scenarios in the project.

REFERENCES

Gary Marchionini, Hermann Maurer.: The roles of digital libraries in teaching and learning Communications of the ACM, Volume 38, Issue 4, April 1995

David Bainbridge, John Thompson, Ian H.Witten: Assembling and enriching digital library collections Proceedings of the 3rd ACM/IEEE-CS joint conference on Digital library ies,2003

Ingo Fromholz, Predrag Knezevic, et al.: Supporting Informa tion Access in Next Generation Digital Library Architec tures In Proceedings of the Sixth Thematic Workshop of the EU Network of Excellence DELOS(2004)

Alexa T.McCray., Marie E. Gallagher.: Principles for Digital Library development. Communications of the ACM Vol ume 44, Issue 5(May 2001)

Rothenberg, J. Avoiding Technological Quicksand: Finding a Viable Technical Foundation for Digital Preservation. Rep. to Council on Library and Information Resources, Jan. 1999

J. Gray. Why do computers stop and what can be done about it? In Proc. 5th Symp. on Reliability in Distributed Software and Database Systems, Los Angeles, CA, 1986.

IBM Autonomic Computing Initiative http://www.ibm.com/autonomic

The Universal Library Project: http://www.ulib.org