INTERNATIONAL RESEARCH JOURNAL IN ADVANCED ENGINEERING AND TECHNOLOGY (IRJAET) E - ISSN: 2454-4752 P - ISSN : 2454-4744
VOL 3 ISSUE 6 (2017) PAGES 2376 - 2385
RECEIVED : 25.10.2017 PUBLISHED : 09.11.2017

Nov 09, 2017

AUTOMATED DIAGNOSIS OF NETWORK PERFORMING PROBLEM

[1]Mrs. K.V. Sumathi, MCA., M.Phil,

Assistant Professor Department Of Computer Science, Selvamm Arts and Science College (Autonomous), Namakkal.

[2]S.B.Ajai Vignesh ,
M. Phil Scholar, Department Of Computer Science, Selvamm Arts and Science College (Autonomous), Namakkal.

ABSTRACT

Software that performs well in one environment may be unusably slow in another, and determining the root cause is time-consuming and error-prone, even in environments in which all the data may be available. End users have an even more difficult time trying to diagnose system performance, since both software and network problems. Diagnosing performance degradation in distributed systems is a complex and difficult task. The source of performance stalls in a distributed system can be automatically detected and diagnosed with very limited information the dependency graph of data flows through the system, and a few counters common to almost all data processing systems. An automated approach for diagnosing performance stalls in networked systems. Flow Diagnoser requires as little as two bits of information per module to make a diagnosis: one to indicate whether the module is actively processing data, and one to indicate whether the module is waiting on its dependents. Flow Diagnoser is implemented in two distinct environments: an individual host's networking stack, and a distributed streams processing system.

Keywords: Diagnoser, Degradation, Task.

1. INTRODUCTION

Failure diagnosis is one of the major challenges that home users and network administrators face today. The problem is more so because there are so many different components which collaborate to realize a particular service and these components belong to different functional domains as well as physical locations. With increasing number of such services, it is important to design systems which enable easy diagnosis of problems encountered and allow determining the root cause of the failures. To diagnose network problems, this paper proposes a system called ***DYSWIS ("Do you see what I see")***. DYSWIS leverages distributed resources in the network. It treats each node as a potential source of network management information, gathering data about network functionality. The state of the network is observed by topologically dispersed nodes in the network. Each node has its own view of the network.

Multiple views of different parts of the network are aggregated to get an overall view of the network. Failures seen by different nodes in the network are correlated, along with historical failure information. Once a diagnosis node has gathered insights on whether other systems are experiencing similar problems, it then combines this information with local knowledge and tries to estimate root causes. This is done using a rule based system, where rules represent the dependencies among various network components. DYSWIS nodes differ in capability level from basic failure detection and maintenance of failure history records to the ability to invoke a set of standardized or customized network probing tools within the system (e.g., ranging from versions of ping and trace route to more application-specific tools) for specific network and application layer protocols and the ability to learn and track network fault behavior, create and manage diagnostic tests based on dependencies between network components or protocols.

Underlying Model for DYSWIS Approach

A medical diagnosis of a patient by a doctor where the patient experiences certain symptoms of an illness, but the cause of these symptoms must be identified by a trained doctor through a methodology which may involve certain diagnostic tests to isolate or confirm possible causes, in addition to leveraging knowledge .Similarly, to find the root cause of a service failure in multimedia services, it requires an understanding of the network and network components that embody these services and dynamic relationships among the networking components and having the right tools and methodology to find the root cause from the known or observed symptoms (failures). It also involves leveraging knowledge about other failures in the network (past failures) and historical information obtained by conducting diagnostic tests.

Steps in DYSWIS Diagnosis

The DYSWIS approach relies on the peer nodes to determine the root cause of the failure. Upon encountering a failure a node asks its peer nodes if they are also observing the failure. The peer nodes, based on their past experience with the same service or based on a probe, conclude that that failure is local to the node. In some cases, the failure can be local to a subnet, access switch, access point or the domain. In other words, locality of failure can extend from node itself to the entire domain. The diagnosis infrastructure may request multiple peer nodes about a particular service to localize the problem.

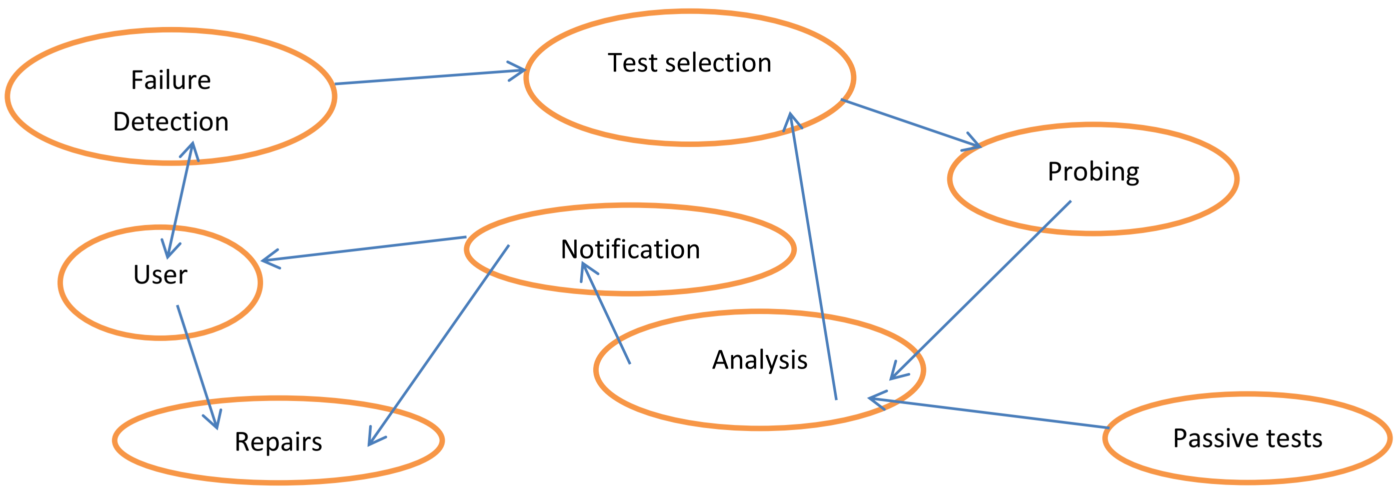

Figure.1. Flow diagram of diagnostic process

The architecture of the proposed fault diagnosis framework consists of the following major functional components: Detection infrastructure and reporting of failures, pre-diagnostic processing and diagnostic test selection and finally diagnostic tests, result analysis and storing historical results. The first step in diagnostic process is detection and reporting of failures. The fault diagnosis framework reports user detected and automatically programmatically) detected failures for analysis to the fault diagnosis system. The failure reports include detailed context information about the failure, such as, one hop distance at different OSI layers, e.g., access point or switch information at layer 2, default router or subnet information at layer 3, first hop SIP proxy server at application layer, timestamp when failure is observed, participating node's hostname and IP addresses.

Example Diagnosis Flow

To explain how our DYSWIS system works, consider a VoIP system (Figure 4). A failure seen by a user, e.g., a call set up failure, can be because of access network failure, mis behaving NAT or failure on SIP proxy server or STUN server authentication failure. Each of
these failures could be caused by mis-configuration, software bugs, server or network overloading or other transient problems in the network. Additionally, there is a complete set of supporting services in the network such as DHCP, ARP, and DNS.

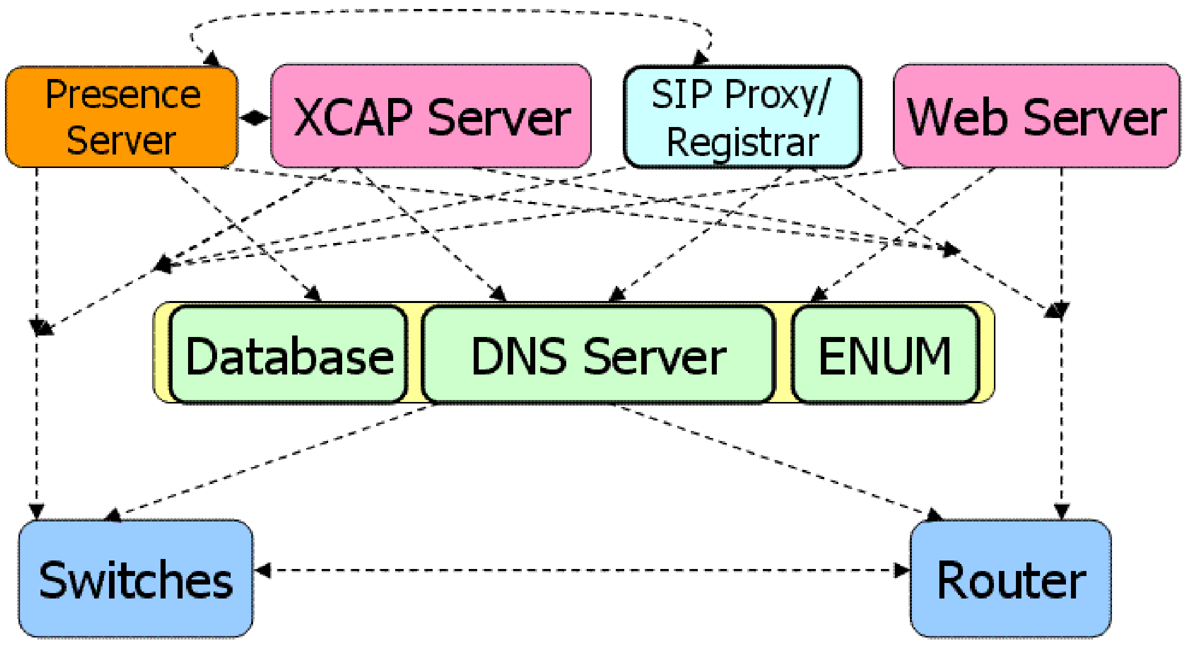

Figure.2. Different network components interacting

Consider a user alice@example.com tries to make a call to another user bob@destination.com and the call does not go through. The end point which tried to make
a call to the remote end point triggers the DYSWIS system to perform diagnosis which takes the following steps:

- The diagnosis node queries if any other node from caller's location has made a call to the user bob@destination.com. In this case, the location could mean from the same subnet, VLAN, access switch/access point or domain.

- The response can be that other nodes have recently made a call to same destination address or to the destination domain, not to same destination address or no node have made any call to destination address or destination domain. It should be noted that historical success or failure information is queried taking into account location of observed failure (hence using the topology) along with functional dependencies.
- Based on the response, the diagnosis node may request another node to send a SIP OPTION message to the destination address or it may request to make a call to the test node in the

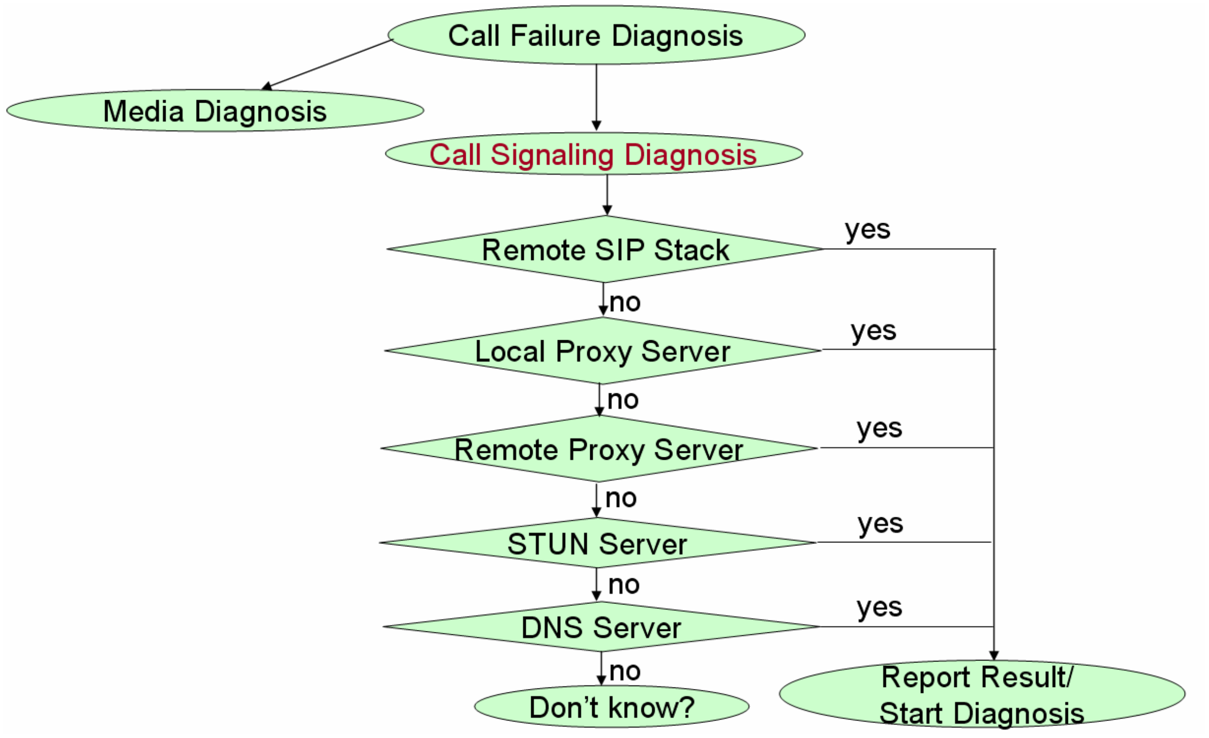

Figure.3. Call failure diagnosis

The call failure diagnosis involves call signaling diagnosis which in turn involves testing of remote SIP end point, local proxy server, STUN server and DNS server. These may further trigger diagnosis of network connectivity and availability of supporting protocols (services). The reports are stored and used to decide the order of queries for future failures. For example, if a call failure to the same destination is reported and diagnosis was already done for that failure, no more tests will be done.

Even well-written software such as the Apache web server can experience sudden spikes in request latency due to head-of-line blocking for disk accesses, contending for shared resources,disk writes, or database queries. Whether these stalls in progress are due to bugs, inefficient locking mechanisms, or calls to backend database servers, they prevent the application software from responding to requests in a timely manner. Another common source of performance stalls is network congestion. A 2011 study of user-facing network traffic at two Google data centers found that packet loss and retransmissions are fairly common: 2.5–5.6% of all user-facing TCP connections retransmit packets.

While fast retransmit and selective acknowledgments (SACK) can avoid complete throughput stalls when packet loss occurs, the study also showed that roughly 1% of all connections stalled for at least

200 ms due to a retransmission timeout (RTO). Even when TCP is able to avoid retransmission timeouts, any retransmissions are costly: short web requests take on average 7–10 times as long to complete when the TCP connection retransmits any packetsIn modern distributed applications, seemingly rare events can have a significant effect on response time. When applications depend on dozens or hundreds of separate services to respond in a timely manner, the outliers in the long tail of the latency distribution are not so uncommon. Amazon e-commerce applications can consult up to 150 services to respond to one request, with each transaction potentially experiencing low throughput or a transient stall due to network congestion, server processing, contention for disk I/O, a longer-than-usual database query, or a transient problem in the network.

Each service is required to meet a service level agreement (SLA), typically to complete 99.9% of transactions in under 300 ms. Even assuming these 150 transactions are perfectly parallel, only 86% of requests will be completed within 300 ms;1 serialized transactions increase delay.

Stalls are hard to diagnose

Many systems exist for monitoring and analyzing the performance of distributed applications. Some require invasive changes to instrument software source code and track individual messages as they are sent throughout the system. While this can help developers and operators to track down subtle bugs and performance problems, the required code changes create a high barrier to entry, especially when monitoring a third-party system for which no source code is available. Other approaches analyze per-packet network captures to try to infer the states of important system elements.

While packet captures can be taken without affecting the performance or source code of the monitored system, they are too expensive to run and analyze continuously, and by nature have little information when a system stops transmitting data. When traffic ceases, it could be that the software has stalled, every transport-layer (TCP) connection has detected network congestion and backed off its retransmissions, or the system has completed all of its current work. Without monitoring the end hosts, it is difficult to reliably distinguish between cause and effect.

Many sophisticated monitors aggregate data from throughout the network to detect systemic problems. These systems are able to locate and detect a wide range of network, software, and system misbehaviors, but mostly rely on complicated analyses that are difficult to recreate, and are of little use for a single host or end user. Another common approach is to perform protocol-specific analysis to detect performance problems exhibited by specific network technologies. These analyses can be invaluable for tracking down difficult and nuanced problems in modern systems of systems. However, applying these protocol-specific insights to new problem domains is not straightforward.

Goals

The goal of this research is to create an approach to messaging performance diagnosis that is efficient enough to run constantly, can automatically detect and report performance stalls using as little information as possible, and is general enough to apply across application domains. It also will enable the following:

- An end user will be able to tell whether their web browser, network connection, or a single TCP stream is causing their performance problems.
- Individual hosts in a distributed system will be able to detect software, connection-specific, or more widespread network problems and report them manner to a monitoring service for cross-correlation and analysis. Such reports will also provide evidence to help pinpoint the root cause of the stall, such as a faulty network interface.
- System administrators and developers will be able to monitor the health of communication in a distributed system, to find which processes or subsystems are preventing progress overall.
- Subject matter experts will apply the basic principles of the Flow Diagnoser approach to finding performance stalls in their own systems.

2. OVERVIEW

Flow Diagnoser approach for locating the source of performance stalls in distributed systems. Flow Diagnoser first constructs a dependency graph, a directed graph that represents the movement of messages between modules of the system. Rather than trace specific messages to see where they are getting dropped or hung up, Flow Diagnoser periodically monitors a few basic counters exported by each node, and performs an abstract analysis of the modules' behavior to make a diagnosis. Once Flow Diagnoser has constructed the dependency graph, diagnosis proceeds in three steps:

1. Periodically snapshot the message counters from each module.
2. Use the counters to infer the module's (in)activity state.
3. Perform a dependency analysis, relating one module's state to that of its dependents and neighbors, to determine whether the module is misbehaving.

The resulting diagnosis is a set of annotations applied to the original graph, with each module labeled to indicate whether it was healthy, blocked by another module, stalled and preventing other modules' progress, or its performance can safely be ignored. In addition to the automated diagnosis, Flow Diagnoser provides several visualizations and summary reports which explain which modules were behaving well, which ones stalled progress, and show the changes in counter values over time. These reports and visualizations also help an expert user to determine if the diagnosis was correct, given the how the counters in the system change over time.

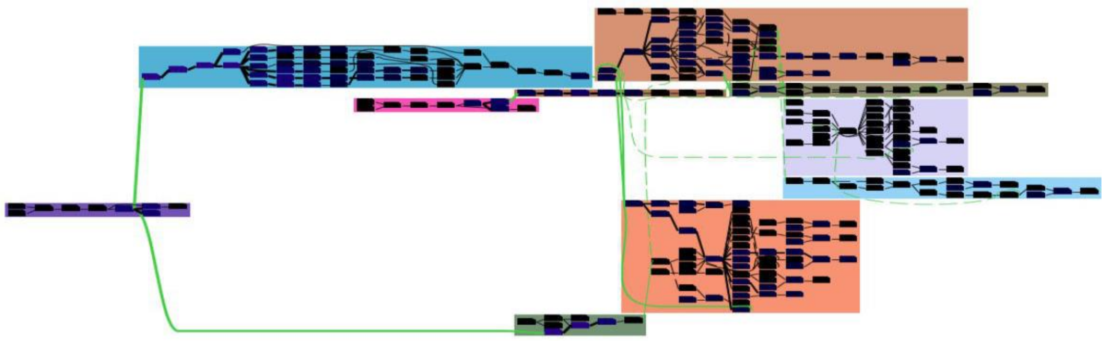

Figure.4. Streams Application showing the flow of messages between processes; a logical view is also available.

Synthetic benchmarks and instrumentation of real applications show that Streams Diagnoser is 93% accurate in attributing the source of performance stalls lasting more than two snapshot periods. As

Flow Diagnoser monitors a system over time, it develops a series of diagnosis results which are assigned to each module in the system. a low-cost, general approach for detecting and diagnosing transient performance stalls in networked and distributed applications. This approach is:

- Automatic and requires no user intervention
- Efficient as it relies only on commonly available counters, with little access to historical data.
- Accurate at diagnosing the source of transient performance stalls before they result in higher-level timeouts.
- General : it is useful for detecting performance stalls in both an end host's networking stack and modern streams-processing systems. Flow Diagnoser is the first performance diagnosis system that provides a general, automated approach that applies to both network-related performance and distributed system messaging, and specifies the minimum amount of information required for diagnosis.

The Flow Diagnoser Approach

Flow Diagnoser approach to finding performance stalls in networked and distributed systems. It consists of three parts,

1. Obtain the dependency graph which describes the movement of messages through the system
2. Periodically snapshot counters for each module in the graph to determine each module's behavior.
3. After each snapshot, perform a dependency analysis over the graph and counters to diagnose performance problems.

The Dataflow and Dependency Graphs

In the Flow Diagnoser model, a system can be viewed as a dataflow graph, where nodes represent modules that process messages, and directed edges identify flows of messages between modules. Each edge is a lossless, finite-capacity pipe with exactly one module at each end. Each module has a finite work queue of messages that it must process; during processing it may transmit messages to other modules. Messages enter the system via sources.

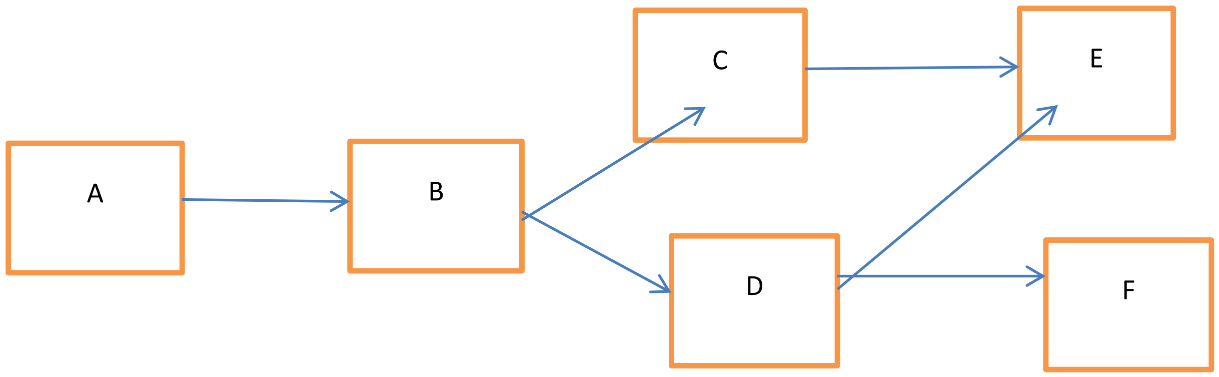

Figure.5. Example dependency graph.

A depends on B for messaging service, which in turn depends on C and D. A system may be either push-oriented or pull-oriented. In the former, dataflow is driven by the source modules. A source module A connected to module B will produce messages and attempt to write them to the pipe that connects it to B. Since this pipe has finite capacity, A's write may block; in this case, B is requiredto read messages from the pipe (depositing them into its own work queue) before A can write further messages. In a push-oriented system, if sources are not producing messages, then the system is idle, but this is not necessarily a problem. In pull oriented systems, dataflow is initiated by sinks. A sink module A connected to B will try to read messages from the pipe that connects the two. If B fails to produce data for A, then A will block. In a pull-oriented system, if sinks are not trying to read messages, then they have no need of data so it need not be provided.

Module Counters

Once Flow Diagnoser has derived the dependency graph, it diagnoses the system's behavior by periodically snapshotting (up to) three counters associated with each module, total_msgs counts the cumulative number of messages that a module has processed (and thus it increases monotonically);

- wait_time counts the cumulative time (increasing monotonically) a module has spent blocked waiting on its dependents to produce a message for it to read (in a pull-oriented system) or to consume messages it has produced (in a push-oriented system);
- queued_msgs tracks the length of the module's work queue.

Flow Diagnoser uses these counters to determine two pieces of information:

1. Whether a module is actively processing messages, and
2. Whether a module attempted to process messages (or has something to do)

For total_msgs and wait_time counters, Flow Diagnoser considers the difference between the current snapshot's value and the prior snapshot's value, denoted as #total_msgs and #wait_time. These differences indicate whether the module was active (a nonzero #total_msgs) or was blocked waiting for service (a nonzero #wait_time). It uses the current snapshot value of the module's work queue (!queued_msgs) to determine whether the module still had work to do when the period ended.

3. LITERATURE REVIEW

The automated diagnosis involves two major steps: (a) Image classification and (b) Image segmentation. Image classification is the technique of categorizing the abnormal images into different groups based on some similarity measure. The accuracy of this abnormality detection technique must be significantly high since the treatment planning is based on this identification. The second step is image segmentation which is used to extract the abnormal portion necessary for volumetric analysis. This volumetric analysis determines the effect of the treatment on the patient which can be judged from the extracted size and shape of the abnormal portion. Many research papers with different approaches for image classification and segmentation are reported in the literature. This chapter provides an extensive survey of existing methods for abnormality detection in brain images.

The drawbacks of the existing methods are twofold: (a) lack of high accuracy and (b) slow convergence rate. Since wrong identification leads to fatal results, accuracy must be exceedingly high in classification and segmentation techniques. Also, these techniques must possess a faster convergence rate which will make them practically feasible for real-time applications. These problems can be overcome by using artificial intelligence techniques and by performing suitable modifications on the existing conventional algorithms.

LITERATURE SURVEY ON IMAGE PRE-PROCESSING

Image pre-processing is one of the preliminary steps which are highly required to ensure the high accuracy of the subsequent steps. The raw MR images normally consist of many artifacts such as intensity in homogenities, extra cranial tissues, etc. which reduces the overall accuracy. Several researches are reported in the literature to minimize the effects of artifacts in the MR images. An analysis on filtering techniques with Gabor filters for noise reduction is performed by Nicu et al (2000). These primitive methods along with reducing the noise blur the important and detailed structures necessary for subsequent steps.

Chunyan et al (2004) have implemented the colour ray casting method to differentiate the region of interest from the background. But this technique is image dependent and not applicable for gray level images. Expectation Maximization Segmentation (EMS) software package is used by Hayit et al (2006) for image pre-processing. The main advantage of this technique is that it is a fully automatic technique. Diffusion filtering combined with simple non-adaptive intensity thresholding is used by Yong et al (2006) to enhance the region of interest.

CONCLUSION AND FUTURE WORK

In proposed DYSWIS system to automatically diagnose network failures and determine the root cause of failures and presented a reference implementation for a VoIP system. DYSWIS system can be implemented for any kind of network as long as probes can be defined, queries can be implemented and an expert can define the dependency rules based on existing probes and queries. As a part of this work, we came up with requirement for a rule-based language which would meet the goals of a rule language for network diagnosis. Our framework uses SIP event notification framework for sending requests and receiving responses. The initial results were obtained by inducing failures manually and observing how DYSWIS triggers diagnostic processing. DYSWIS diagnoses complex end-user’s network problems using end-user collaboration.

It provide a new framework for collaborative approach and diagnosis strategies for various fault scenarios. We provide a detailed design to discover and communicate with collaborating nodes. Also, provide a framework for administrators and developers to participate to contribute to expand the diagnostic system.

To implemented a prototype of the DYSWIS framework and present how easily the participants add new rules and modules on top of the framework in order to diagnose several common network faults. We set up these scenarios with real network devices and diagnosed them using those rules and modules we have created. While local probing with traditional diagnosis tools fail to point out the cause of these fault scenarios, our evaluation presents that DYSWIS can effectively narrow down the problematic regions and pinpoint the root causes.

FUTURE WORK

In implementation, encoded dependency relationship as rules in the form of queries and probes; however, with networks growing in terms of components, services and protocols, there is a need to generate the dependency relation automatically using statistical mechanisms and using temporal correlation among failures detected. Secondly, need instrumentation of applications to detect and report failures in order to trigger diagnosis. To detect failures without requiring software upgrade would require us to detect network failures using traffic analysis. This in turn would require specifying protocol details using a rule language to the traffic analyzer.

To identified requirement of a rule language for failure diagnosis. One of the tradeoffs in developing a rule language for diagnosis is simplicity vs. capability. An expert must be able to specify rules with ease without requiring much knowledge about a programming language. However, this limits the functionality that can be expressed in a rule. The system needs to provide mapping between functionality of the system vs. the tools available to the expert. Providing a fixed mapping reduces the enhance-ability of the diagnosis system for new probes. A more scripting-based approach gives more flexibility but more complexity to the expert. A system which gives flexibility by taking external binaries/scripts and output of such binaries and scripts back to the rule language as well as provides a fairly high level way of representing knowledge may be good approach, a mix of XML and shell script style. Finally, failure event correlation based on rules is another area of future work.

REFERENCES

1. Binzenhöfer, A., Tutschku, K,. Graben, B., Fiedler, M., Arlos, P., “A P2P-Based Framework for Distributed Network Management”, New Trends in Network Architectures and Services, LNCS, Loveno di Menaggio, Como, Italy, 2006.
2. Miao, K., Schulzrinne, H., Singh, V., Deng, Q., “Distributed Self Fault-Diagnosis for SIP Multimedia Applications”, To appear in MMNS’2007, 10th IFIP/IEEE International Conference on Management of Multimedia and Mobile Networks and Services
3. Rish, I. Brodie, M. Odintsova, N. Sheng Ma Grabarnik, G., “Real-time problem determination in distributed systems using active probing”, Network Operations and Management Symposium, 2004. NOMS 2004. IEEE/IFIP.
4. Utton, P.; Scharf, E., “A fault diagnosis system for the connected home”, Communications Magazine, IEEE, Volume 42, Issue 11, Nov. 2004, 128 - 134.
5. Xu Chen, Ming Zhang, Zhuoqing Morley Mao, and Paramvir Bahl. Automating network application dependency discovery: Experiences, limitations, and new solutions. In OSDI, pages 117–130. USENIX Association, 2008.
6. Yu-Chung Cheng, Mikhail Afanasyev, Patrick Verkaik, P´eter Benk¨o, Jennifer Chiang, Alex C. Snoeren, Stefan Savage, and Geoffrey M. Voelker. Automating cross-layer diagnosis of enterprise wireless networks. In SIGCOMM, pages 25– 36, 2007.
7. Y. Zhang, Z. M. Mao, and M. Zhang, “Effective diagnosis of routing disruptions from end systems.” in Proc. of NSDI, San Francisco, CA, USA, April 2008.