ASSESSING OPERATIONAL IMPACT IN ENTERPRISE SYSTEMS WITH DEPENDENCY DISCOVERY AND USAGE MINING

A Dissertation
Presented to
The Academic Faculty

by

Mark Bomi Moss

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2009

ASSESSING OPERATIONAL IMPACT IN ENTERPRISE SYSTEMS WITH DEPENDENCY DISCOVERY AND USAGE MINING

Approved by:

Dr. Calton Pu, Advisor
College of Computing
Georgia Institute of Technology

Dr. Leo Mark
College of Computing
Georgia Institute of Technology

Dr. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Henry Owen
Department of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Ling Liu
College of Computing
Georgia Institute of Technology

Date Approved: April 9, 2009

To my loving parents

Henry Clay Moss, Mary Daniel

and

Daniel & Ninfa Cuevas

ACKNOWLEDGEMENTS

I would like to thank some of the people who have been especially generous in gracing me with their wisdom, technical expertise, time, and support. First and foremost, Calton Pu has been my advisor and mentor, and has always given me the benefit of his extensive experience in research, publishing, and many other facets of scholarship at this level. Calton was extremely patient, gave me a tremendous amount of freedom to explore my passions, while also sharing timely insights to help keep me motivated and on track. I was also blessed with the experience and insights shared by the other member of my committee: Mustaque Ahamad, Ling Liu, Leo Mark, and Henry Owen. I would also like to thank Susie McClain, Deborah Warren, Andrea Barrow, and many of the other members of the College of Computing Graduate Staff.

Warren Matthews of the Georgia Tech Research Networks Operations Center (GT-RNOC) was incredibly supportive of my research, and he helped me access the treasure trove of Georgia Tech campus-wide systems and network data I needed for my experimentation. In a similar vein, I must also thank Russ Clark and Mohammed Mansour for their technical support: Russ, for his assistance in the small-scale research conducted in the Center for Experimental Research in Computer Systems (CERCS) lab; and Mohammed, for coordinating access to various servers and software systems in the initial stage of my investigations. I must also thank Simon Malkowski and Danesh Irani, as fellow graduate students, for their continual support in implementing, testing and analyzing our prototype impact assessment system. I wish both of them the greatest success in their continuing studies. My sincere thanks go to COL Gene Ressler, COL

Barry Shoop, Dr. Jean Blair and LTC Greg Conti, for your continued support, encouragement and faith in my ability to achieve this goal. The United States Army, through its' Advanced Civil Schooling Program, has offered me this wonderful opportunity: to be able to serve my country while studying this discipline of Computer Science, for which I am truly grateful. I offer my sincere thanks to all of the personnel who support the Army's ACS Program. I look forward to my continuing journey, and to sharing my knowledge, experience and passion with my students, and fellow instructors and researchers. To all who have helped me on this journey, and especially those listed above, I wish you success and Godspeed in your future endeavors.

Finally, I must thank my two angels – my daughter Danielle, and my son Matthew – for your love, support and patience, especially on those days when I was thoroughly absorbed with research. And to my wife, Sylvia – you have been my friend, confidant, cheerleader, and coach. You have always expressed a deep faith in my abilities to climb this mountain, and have always been there when I needed a warm shoulder of support. I love you, and I am proud and extremely blessed to share my success and my life with you, and with our family.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

CERCS	Center for Experimental Research in Computer Systems
CPR	Campus-Wide Network Performance Monitoring and Recovery
DNS	Domain Naming System
DoD	Department of Defense
GT	Georgia Institute of Technology
ICMP	Internet Control Message Protocol
IP	Internet Protocol
IT	Information Technology
J2EE	Java 2 Enterprise Edition
MAC	Mission Assurance Category
MTTR	Mean Time To Repair
RNOC	Research Network Operations Center
SLA	Service Level Agreement
SNMP	Simple Network Management Protocol
WEKA	Waikato Environment for Knowledge Analysis

SUMMARY

Enterprise systems are growing larger, more distributed, and increasingly complex. They can be composed of hundreds or thousands of heterogeneous workstations and servers, connected via various networking devices, which allow business users to access critical data via multi-tier applications and web services. They can vary in architecture, available bandwidth, computing power, and the amount of black-box resources employed. System administrators are often required to assess the impact on business operations when an enterprise system component fails, which we refer to as ***assessing the operational impact***. Operational impacts can also be caused inadvertently when enterprise system components are reconfigured. Assessing operational impacts accurately is critical to providing business executives with information needed to allocate limited Information Technology resources optimally – for example, maintenance personnel, time, and dollars.

We claim that assessing operational impact requires that administrators relate the component failure to the affected users in a manner that is clear and understandable by business executives. A number of approaches have been presented to calculate these kinds of impact, but many of these approaches have focused on the calculating the dependencies at the application & infrastructure levels. The applications are important only in that they provide a means for the business users to access their critical business data stored in files, databases and other (possibly remote) repositories, or to contact other users directly in a timely manner. Furthermore, the importance of different sets data will vary over time. For example, a certain set of financial data, and the ability to access and

modify this data, might be significantly more critical to the business operations near the end of the fiscal year as opposed to other times. Consequently, to more accurately determine the operational impact, an impact assessment system must also monitor the various data sources accessed, the various applications used to access them, and the periods of time for which accessing these files are truly critical to the business users.

This paper presents a framework for monitoring the dependencies between users, applications, and other system components, combined with the actual access times and frequencies. We use operating system commands to extract information from the end-user workstations about the dependencies between system components. We also record the times that system components are accessed, and use data mining tools to detect usage patterns. This information can then be used to predict whether or not the failure of a component will cause an impact during certain time periods. Furthermore, we designed this framework to require minimal installation and management overhead, and to consume minimal system resources, so that it can be employed on a variety of enterprise systems, including those with low-bandwidth and partial-connectivity characteristics. Finally, we implemented this framework in a test environment to demonstrate the feasibility of this approach. This combination of understanding how and when users access various system components allows us to better assess current and future operational impacts.

CHAPTER 1

INTRODUCTION

1.1 Problem Description

System administrators are often required to assess the impact on business operations when a component fails, which we will refer to as ***assessing the operational impact***. Presenting this assessment to management executives requires that we make the impact relevant to the operations of the business. To say "Our application server has failed" is to state a technical event. This may be clearly understood by the Information Technology (IT) staff, but will probably not have much significance for the executives. A more relevant statement for them would be "We will not be able to access our purchasing applications for the next two hours because our application server has failed", where you have connected the technical event clearly to one or more business operation impacts. Most users outside of the IT staff understand how automated systems affect their business only in terms of the applications that they use on a regular basis; therefore, assessing and presenting the operational impact effectively requires that we be able to clearly relate lower-level technical events to understandable concepts like user-level applications.

The timing of the events is also important in assessing operational impact. If none of the employees has a need for the purchasing applications at the time of the assessment, then there is arguably no significant operational impact. In some cases, it might be practically impossible to guarantee that there won't be at least one employee trying to access the purchasing application; still, we should also develop some

understanding of when an application is actually needed in order to more accurately assess the impact. This is also important to minimize the number of false reports. It is our experience that an overabundance of false reports can erode management's confidence in the validity of these assessments as much as not reporting valid impact assessments.

Consequently, to accurately and effectively assess operational impact, we must (1) relate the technical events to user-level applications with which executives and other users are most familiar, while (2) considering the times that these applications are needed to conduct business operations. These two primary objectives can be complicated by a number of factors, including: the size and complexity of the system we are monitoring; configuration changes to that system; the variety of application access patterns that might be encountered; and handling black-box components, among others. The following sections address these concerns in greater detail.

Consider the following example in Figure 1, where various users, applications, computers and supporting infrastructure devices (e.g. routers) are shown. The users at computer A use the Internet Explorer application to access the remote site. Similarly, the users at computer B use the iCal application to coordinate their schedules with the users at computers C and D.

Now suppose that the router R fails for a 2-hour duration as shown in Figure 2. The users at computer A might be unable to access the remote website during the router outage. Similarly, while the users at computer B should still be able to coordinate their actions with the users at computer C, they might be unable to contact (and coordinate with) the users at computer D.

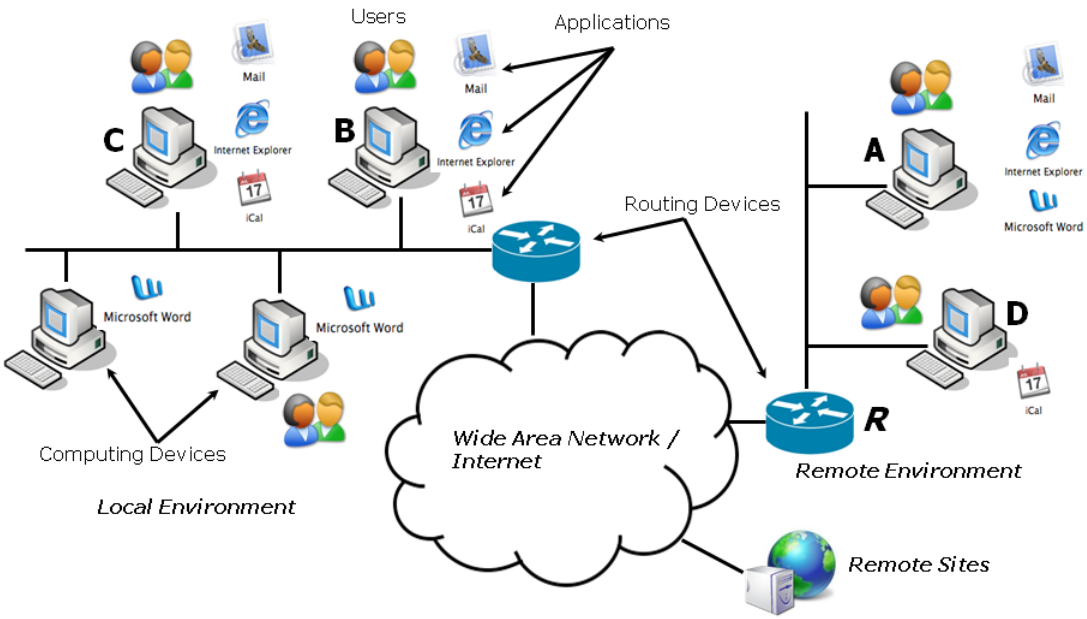

Figure 1 - Sample Enterprise System

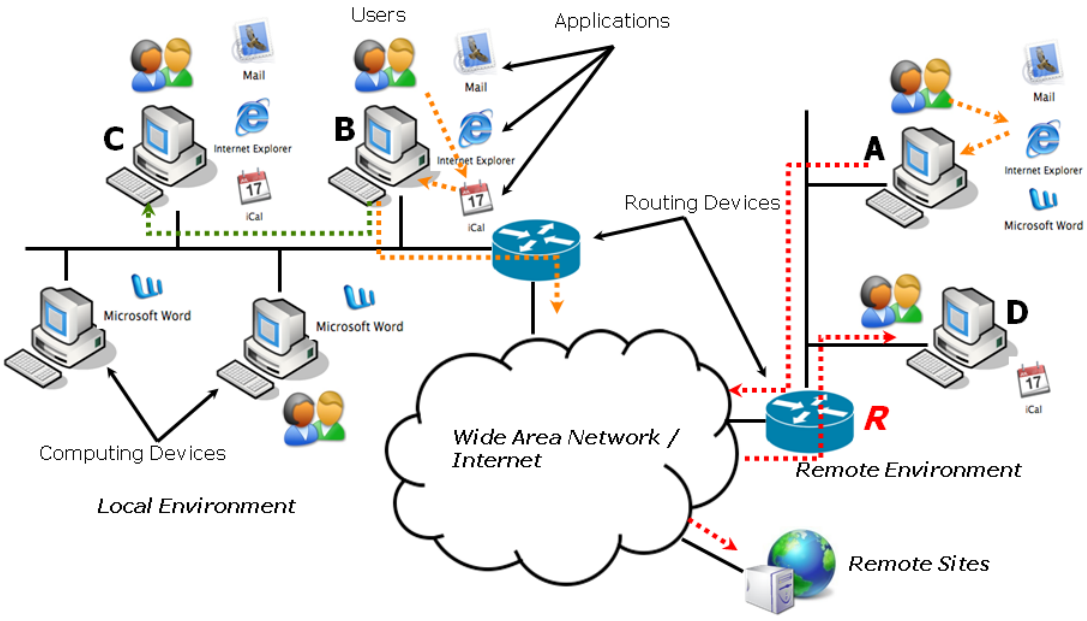

Figure 2 - Operational Impact of Router Failure

Furthermore, if the users at computers A and B did not actually need any of the affected services during the outage period, then there was (by our definition) really no adverse operational impact. There will normally be an "operational support" requirement for IT staff technicians, engineers and administrators to repair the damage and restore systems and services to their normal status, but to significant impact on the other business operations. Along these same lines, we can also use some of these principles to seek optimal times for minimizing future operational impact. For example, instead of encountering an unplanned/unexpected router outage, the IT staff might be aware of a need to apply a critical security patch to the router firmware, and or upgrade some of the hardware components.

Assessing operational impact is topical and relevant to managing enterprise systems in many environments. Mamaghani examines the role of Information Technology (IT) in supporting enterprise business operations [1]. He gives a chronological description of how business managers have viewed the effects of IT on their operations. The original focus in the 1980's was "*Are my business and IT infrastructure performing?*" Economic pressures and the fast pace of business changes forced managers to develop their focus further, and include concepts such as: effective *linkage of technology and end-to-end operations*; total *visibility* over business processes; and, *disadvantaged situation prediction* and prevention. Our research focuses on this combination of concepts. Operational impact assessments are intended to provide business managers with sufficient visibility to determine when technology failures in their enterprise system will have a significant impact on their business operations. Furthermore, we integrate timing and usage data to permit managers to predict the

potential impacts of failures, instead of being forced to wait until the failures actually occur. Managers must receive clear, accurate and timely operational impact assessments in order to run their businesses effectively. A recent New York Times article reports on the impact that web site crashes have on individuals; and, with the growing number of online services, on businesses as well [2]. The article discusses a website that users can use to test if a particular website is inaccessible for everyone, or just for them. This kind of analysis is becoming more critical as developers leverage pre-existing services and components to build ever more complex systems. This further confirms the importance of our focus on developing ways to determine the impact on users and complex, higher-level systems when lower-level components fail. There are similar concerns and requirements in academic and military environments as well.

Many military command, control and communications systems are complex, multi-layered compositions of various resources that provide critical services to operational commanders. When a resource event occurs, it is important to alert the commanders to any services that have been adversely affected by this event as quickly as possible, because the lack of these services might have an impact ongoing or upcoming operations. Grimaila, Mills and Fortson document the clear and critical need to link the result of a cyber incident – which may result in infrastructure damage, and/or the compromise of some cyber resource – to the mission impact experienced by all of the affected organizations [3]. They discuss the importance of assessing the potential damage in an accurate and timely manner, as well as the need to distinguish between damage (i.e. technical impact) and mission (i.e. operational) impact. They acknowledge that many organizations neglect to develop and maintain this type of information because

of difficulties in obtaining the raw data, lack of qualified personnel, and/or fear that the information collected could be used by their adversaries to target their most critical and vulnerable assets. They also propose a distributed information asset tracking system designed to identify information dependencies, which uses Host-Based Security System (HBSS) software agents [4]. We concur with Grimaila et. al. on the overall importance of assessing operational impact by determining the relationships between technical events and operational needs. Furthermore, we believe that integrating the data about actual usage patterns will allow the system administrators to more accurately assess the operational impact on the system's operators. And while the HBSS referenced seems to focus on security issues – detecting intrusions, ensuring that software patches have been installed, updating anti-virus signatures on a frequent basis – our systems focus more on the raw components as defined in our Dependency Topology Model.

In fact, a report from Friedman acknowledges that a key objective of the Department of Defense's Critical Infrastructure Protection (DCIP) Program is to allow "military commanders and DoD policy makers to effectively manage the impact of failing infrastructure assets" [5]. Since an attack on one critical asset can impact the operation of larger systems, such as those used for transportation, medical service, logistical support, etc., protecting our critical infrastructure requires that we "operationalize" the DCIP Program. This overall effort will require a number of coordinated actions, including a systematic effort to "identify critical assets and dependencies, and the impact of their degradation or loss."

Calculating *operational impact* involves determining those operator/end-user level services that have been affected by one or more resource events, which typically

range from complete failure to significantly degraded performance. Knowledge of the affected services can then be used to alert the operators, and determine how to prioritize repair resources. This is similar to the general problem of "root cause analysis", where the goal is to determine the main cause of one or more service or application failures. Both problems generally require some knowledge of the system component dependencies; however, calculating operational impact is more like a reversal of the root cause analysis problem.

We must also understand how employees use enterprise system components in order to link the technology to the end-to-end business operations. Older, traditional views of enterprise systems included desktops connected to servers via local area networks. Given the choice between monitoring the employee-component interactions from the desktop end-systems, or from the servers, we choose to monitor from the end-systems. In this way, we capture the interactions between an end-system and remote servers, as well as the local interactions that involve only components located on the end-system itself. Also, enterprise system capabilities have also increased over time. Modern enterprise systems support telecommuting, globalization, and outsourcing. Shan's analysis indicates that future employees will continue to operate from remote locations via electronic means including laptop computers, Personal Digital Assistants (PDAs) and other mobile devices [6]. This further supports an end-system based approach, which allows monitoring of end-system local interactions during periods of intermittent or low-bandwidth connectivity. This is also relevant in certain military environments, where deployed units operate over low-bandwidth, high-latency links (i.e. via satellite

communications), and operational demands require them to move frequently, causing them to break and reestablish their communication channels on a regular basis.

1.2 Thesis Statement

We can integrate dependency discovery techniques with the data received from mining usage patterns to allow the administrators of enterprise systems to more easily identify and assess the likelihood that a given technical event will cause an operational impact.

1.3 Organization

The rest of this dissertation is organized as follows:

- **Chapter 2** clarifies the problem further, defines some of the key terms, and addresses some of the challenges faced when assessing operational impact.
- **Chapter 3** examines some of the previous research related to this problem, and demonstrates where and how our work diverges from previous efforts.
- **Chapter 4** discusses the dataflow phases of our impact assessment process, with a higher-level emphasis on the how the raw data is processed to extract topology and usage information.
- **Chapter 5** examines and compares centralized, partially-distributed and fully-distributed methods for processing impact assessment.
- **Chapter 6** discusses the technical details of our impact assessment system, with a lower-level emphasis on the key algorithms used when implementing the prototype.
- **Chapter 7** analyzes our experimental results, including our tests of the smaller-scale data, the large-scale data, the centralized and distributed comparisons, and clustering data.

- **Chapter 8** summarizes our conclusions, and presents some suggestions for future research.

CHAPTER 2

DEFINITIONS AND CHALLENGES

2.1 Problem Statement, Goals, and Non-Goals

Our goal is to present a framework that helps system administrators assess the operational impact by determining the users affected by a component failure. This framework supports assessments in the current time period, and also provides a predictive capability by leveraging the information generated from usage pattern mining to infer the likelihood of impacts during future time periods. We don't expect this approach to assess the operational impact perfectly; the intent is that it will provide clear, operationally focused, and timely feedback that assists system administrators in assessing the operational impact for the executive users of the system. Our approach is based on collecting operating system data from selected end-systems to construct a model of the intra-system and inter-system resource dependencies. This information is then aggregated to construct a dependency model for the overall enterprise system. The data is also time-stamped, and data mining techniques are applied to detect usage patterns. The dependency topology and usage pattern information is then used to assess operational impacts.

The aim of our project is to create tools and methodologies to enable administrators to better assess the operational impact of a resource failure. We want our tools to:

- Assemble a model that captures the resource and resource dependency information clearly. Our model is represented using standard directed graph notation.
- Determine the users potentially affected by a resource failure. This is done by using the pair-wise dependencies contained in our model to calculate the transitive dependencies from the failed resource.
- Filter the users most likely affected during the designated time window. The specific timing data and detected usage patterns should be used to predict those users that are likely to be actively using the affected resources. If the model is continually updated, then the results for users affected at (or very near to) the actual time of the resource failure will be more accurate.
- Provide the impact assessment for a given resource failure in a reasonable amount of time. The impact assessment is needed by executive system users to make critical decisions in a timely manner, possibly as part of a larger legal or regulatory requirement. For example, if certain military users or resources have been affected, the senior commander may be required to cancel an operation, or take other emergency actions, to avoid loss of life. In a different example, the loss of routing capability might require a Network Service Provider to notify their affected customers within a certain period of time per an established Service Level Agreement. In our experience, this amount of time can be measured in hours or even minutes.

The requirements for our tools from an installation and management perspective are defined here. The amount of effort needed to install, configure and manage a set of tools can affect the decision to implement those tools. We believe that tools requiring

application-specific configurations and/or modifications are less likely to be implemented for concerns (real or perceived) that the investments needed to implement the tools will outweigh the benefits gained. Consequently, our tools should:

- Require minimal application-specific knowledge and configuration, especially where administrators are required perform the configuration manually.
- Require minimal modifications to applications, middleware or other system components.
- Not significantly perturb system performance.

It is also important to state some of the non-goals for our tools. It is our belief that truly assessing operational impact requires both a technical understanding of the system, and a solid understanding of how the users utilize the system's resources to achieve their business objectives: commercial, military or otherwise. The intent of our tools is to provide topology and usage information that complements the administrator's normally technical perspective of the system, and allows the administrators and executive system users to more accurately collaborate in assessing the impact. Consequently, our tools are not intended to replace administrators and/or executive users; rather, they are designed to assist them with enterprise systems that are increasing in size and complexity. Also, since our tools are not intended to operate autonomously (e.g. directing system configuration changes without administrator supervision) we feel that our tools do not have to be foolproof: they should be robust, and should minimize false positive and false negative assessments as much as possible.

Many of these large-scale systems also contain hundreds, and possibly thousands, of cooperating end-systems. We believe it is feasible to leverage the idle computing

cycles, free disk space and network bandwidth of these end-systems for data collection. Our intent is to aggregate the impact assessment data collected at the end-systems to provide sufficiently accurate impact assessments for the overall system. Distributing the impact assessment workload of our tools across the end-systems might improve the scalability of our approach, but this is not the only reason for using this approach. The users interact with a system's resources via the end-systems. We believe that monitoring system usage patterns at the end-system level is efficient in that it avoids transmitting that data across the network to some other collection point. Also, it offers potentially more complete and efficient coverage of end-system (local-only) operations, as opposed to monitoring the local operations for potentially hundreds of end-systems from a remote location.

2.2 Key Terms, Concepts and Definitions

We define an *Enterprise System* as a distributed system of components that are used in combination in pursuit of one or more functional objectives. We model a distributed system as a graph of communicating resources. In our model, the nodes of the graph are resources, and the edges of the graph represent the dependencies between resources. A directed edge from resource A to resource B means the resource A is in some way dependent on resource B. We discuss the specific resource and dependency definitions in a later chapter. Consequently, we model an enterprise system as a directed graph of its' distributed resources, where the nodes represent the system's resources, and the edges represent the functional dependencies between resources. An edge from a source node to

a sink node implies that the failure of the sink node would likely prevent the source node from completing its tasks successfully. Our terminology and definitions are shown here:

$$\boldsymbol{Enterprise\ System} = \langle Resources, Dependencies \rangle$$
$$where\ Dependencies = \{d_{i,j} | r_i, r_j \in Resources\ and\ r_i \rightarrow r_j\},$$
$$source(d_{i,j}) = r_i, sink(d_{i,j}) = r_j\ and\ RealUsers \subset Resources$$

$$\boldsymbol{Technical\ Event} = \langle Failed, t_{failure}, duration, Status \rangle$$
$$where\ Failed \subset Resources,$$
$$Status = \{d: active(d, t_{failure}) | d \in Dependencies\},$$
$$and\ active(d,t) = \{1\ if\ d\ occurs\ at\ time\ t; 0\ otherwise\}$$

$$\boldsymbol{Impact\ Assessment} = \{\langle Path, t_{start}, t_{stop}, p_{impact} \rangle\}$$
$$where\ Path = \{d_{(1)}, d_{(2)}, \dots, d_{(k)}\}, \forall i\ sink(d_{(i)}) = source(d_{(i+1)})$$
$$source(d_{(1)}) \in RealUsers, sink(d_{(k)}) \in Failed,$$
$$and\ t_{failure} \preccurlyeq t_{start} \preccurlyeq t_{stop} \preccurlyeq (t_{failure} + duration)$$

We define a *Technical Event* as a 4-tuple which represents the instance where a certain set of resources have *Failed* at time $t_{failure}$, and will not be repaired or restored until $(t_{failure} + duration)$. In most cases, the average repair time (i.e. MTTR) can be used as an approximate duration value. *Status* captures the operational status of the system resources at the time of failure. Capturing all system status data might not be possible in some environments, but even partial status data can be useful in assessing impact. We then define an operational *Impact Assessment* as a set of 4-tuples. Each tuple represents how one user will be affected by one of the failed resources along a given *Path*, during the period from t_{start} to t_{stop}, with a likelihood of p_{impact}. The path information is generated from the topology data, while the t_{start}, t_{stop} and p_{impact} values are generated from the usage pattern data.

2.3 Challenges, Assumptions and Constraints

2.3.1 System Size and Complexity

Enterprise systems are growing larger, more distributed, and increasingly complex. They can be composed of hundreds or thousands of heterogeneous workstations and servers, connected via various networking devices, which allow business users to access critical files via multi-tier applications and web services. They vary in architecture and composition: some are composed of fixed-location components, like rack-mounted servers, continuously connected by high-bandwidth links. Others are composed of smaller, mobile components – for example, vehicular-mounted systems – connected by lower-bandwidth links with partial connectivity. The variance in these characteristics can make it very difficult to monitor and manage these types of systems.

End users normally interact with a system via desktop and laptop workstations. Also, even with the popularity of network-based file storage and thin client computing, many users still execute applications and access files on their local workstation, as well as accessing network-based services and data. For these reasons, we argue that an impact assessment system must monitor both workstation-local and network-based activities to develop accurate impact assessments. As the number of workstations increases, this can affect the way monitoring data is collected – remote data collection might be problematic, especially in cases with low bandwidth and/or partially connected links.

2.3.2 Configuration Changes

Most enterprise systems also require configurations changes for a variety of reasons. Security concerns might require that a patch be applied to specific operating system or

applications components to prevent a recently discovered vulnerability from being exploited. A series of ports might be blocked at a firewall to prevent infection by a rapidly spreading virus. New servers and software might be installed and configured to support new and/or increased capabilities, thus requiring modification to the underlying network to support the increased data flow. A merger of two companies might require that the separate business systems and networks be integrated to support the resulting corporation.

In each of these cases, the impact assessment for certain users can be affected by changes to the configuration. Configuration management and monitoring are still difficult problems, often complicated by the lack of trained administrators and time needed to properly document the changes. Since manually updating impact assessments would be correspondingly difficult, an impact assessment should provide automatic monitoring of changes where possible to ensure that the assessments remain accurate when enterprise system components are added, reconfigured and/or removed.

2.3.3 Timing & User Access

Timing is a very relevant factor in assessing operational impact, since the importance of a particular set of data often varies over time. For example, consider an employee tasked to prepare a daily report for each morning at 8:00 A.M. In order to prepare the report successfully, that employee will need to access the input information between 6:00 and 7:45 A.M., which is located on a remote file server. If a router failure at 6:15 A.M. prevents the employee from accessing the needed information, then the “router failure event” has caused the operational impact of preventing completion of the morning report. On the other hand, if the router failure occurred at 7:55 A.M. or later, then there would be

little or no operational impact on that day's morning report. Clearly, importance of the report information on the remote file server varies over time. Consequently, the applications used to access this information also vary in importance as well. Since users may not be aware of these variations, automated techniques would be helpful in detecting these timing relationships.

By extending this example, we can also show why using uptime/downtime percentage metrics are not necessarily sufficient for assessing operational impact accurately. Some service providers measure the amount of time that a service has been available during a certain period, divided by the total amount of time in that given period (normally days, weeks or months), as the "uptime" for that service. SLAs and other Quality of Service agreements are then based, at least in part, on these uptime measurements. Consider the remote file server mentioned in the example above. It is possible that file server could have a very high uptime measurement, but constantly fail for relatively small durations during critical times (i.e. morning report preparation). This would cause a significant operational impact in spite of the high uptime measurement. Similarly, suppose the remote file server had to be taken offline on a regular basis for a few hours to perform maintenance, reorganize data for faster access, etc. If care is taken to ensure that the file server is available as required during morning report preparation, then there will not be any significant operational impact, in spite of the low uptime measurement. Uptime measurement is very helpful from a service provider perspective; our intent is to show that this aggregate measurement of performance is not necessarily sufficient to measure operational impact accurately.

We propose that there are at least two distinct ways to address timing information in impact assessments: one way is with respect to fixed dates and times, and the other is relative to the execution of other applications. From a different perspective, certain applications are "scheduled" for execution at certain times, while other applications are executed "on-demand" with respect to other applications or events. With respect to "scheduled" dates and times, certain applications are executed at certain specific times, like the daily morning report mentioned above. As another practical and widespread example, many large organizations have to deal with the concept of a fiscal calendar. At some specific date during the year (i.e. October 1st), the current fiscal year officially ends, and the new fiscal year begins. This normally requires that many financial transactions be completed absolutely no later than the day before. It is a normal occurrence in many of these organizations to see financial officers and other supporting personnel feverishly completing paperwork at 11:50 P.M. on September 30th. The files that are used for these transactions are important all year long, and are normally updated throughout the entire fiscal year to maintain their correctness and consistency. During the last few days before the new fiscal year, however, their importance is significantly increased because of the various legal and regulatory requirements.

On the other hand, some applications are more likely to be executed "on-demand" within a certain time frame after other specific applications have recently been executed. One example is consider applications and data from a workflow perspective: if application X is used to preprocess data for application Y, then the current execution of application X implies an inductive probability that application Y will be executed within the near future. Consider the workflow for a military unit that has just decided to conduct

a large operation. In preparing for this operation, the planners will access a database of previous operations to generate orders for the mission. The operators will use these orders to execute the mission, and produce a status report with the results. Though there is no specific date, as in the fiscal year example above, that can be used as a measurement reference for impact assessment, we can still use relative measurements. Given this example, consider the timeline for planning and executing this mission as shown in Figure 3.

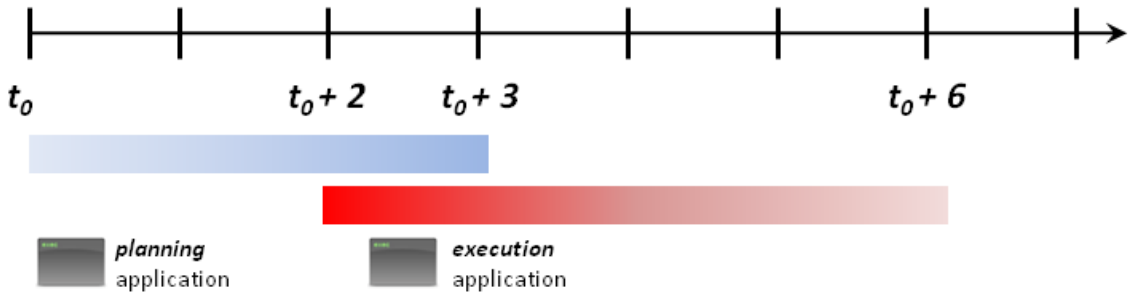

Figure 3 - Timeline for Conducting Missions

Suppose the decision to conduct the mission occurs at time t_0. The planning phase (shown in blue) requires 3 days, and the execution phase (shown in red) requires 4 days. The application used by the planners to generate orders might be most critical (and susceptible to operational impact), between times t_0 and $(t_0 + 3)$ days. Similarly, the applications used to execute the mission might be most critical between $(t_0 + 2)$ days and $(t_0 + 6)$ days. The overlap in criticality from $(t_0 + 2)$ days to $(t_0 + 3)$ days is normal, and accounts for the handoff of the mission orders from the planners to the operators. Given that mission planning began at t_0, we can use the time periods relative to t_0 to assess operational impact. More specifically, if the planning application is active

at t_0, then it is significantly more likely that the execution application will be active between $(t_0 + 2)$ and $(t_0 + 6)$, and especially during the $(t_0 + 2)$ to $(t_0 + 3)$ period. We can use the activity status of the planning application to better determine the likelihood that the execution application will be operationally impacted.

2.3.4 Black-Box Components

Many enterprise systems are structured as large-scale distributed systems, and composed of multiple communicating components. They often employ proprietary code, third-party services, and similarly protected components that prevent the system administrators from gaining complete and in-depth visibility of the entire system. Understanding the structure of such systems can be difficult, especially when they are composed of these "black-box" components. For local components, black-box implies that component's source code is unavailable; for remote components, black-box implies a service hosted by a third-party provider, where visibility of the component's operation is restricted to the interface level.

The black-box nature of these components complicates the monitoring process, but they must still be considered when monitoring the system as a whole. For impact assessment purposes, it is necessary to detect when users and applications are accessing a given component X. Then, to detect transitive dependencies, we should then determine if component X, in turn, accesses other components. If component X is a black-box component, we may not be able to gather directly, or infer with sufficient accuracy, component X's dependency information. A practical example involves using traceroute() to detect the path from a source address (normally the host computer) to a designated target address. In some cases, certain routing devices along the path might have been configured to ignore the requested ping responses, which prevents those device's IP

addresses from being included in the traceroute() output. These kinds of actions will limit our ability to detect potential impacts, thus limiting the overall coverage of our impact assessments.

Our goal is to design tools that help administrators assess the operational impact by determine the users affected by a component failure. These tools should require minimal support from the components themselves, and should avoid assumptions that components will provide specific and continued support for our particular methodology. We don't expect these tools to assess the operational impact perfectly; the intent is that they will provide clear, operationally focused, and timely feedback that assists system administrators in assessing the operational impact for the executive users of the system.

In this paper, we describe a specific approach towards assessing operational impact. Our approach is based on capturing operating system diagnostic data from numerous end-systems in an application-independent, passive manner. The captured data is used to construct a model of the intra-system and inter-system resource dependencies for each end-system, and is then aggregated across all of the end-systems to construct a dependency model for the overall system. The data is also time stamped, and data mining techniques are applied to detect resource usage patterns. We show that the captured data is sufficient to detect key resources and resource dependencies, and that the time stamping allows us to determine if an actual operational impact occurred with reasonable success. The use of passive tracing, and avoidance of application or middleware modification, reduces the intrusiveness of our approach, making it more widely applicable to a variety of distributed systems.

CHAPTER 3

RELATED WORK

There has been significant research on impact analysis, and translating the effects of technical actions into business-relevant effects. Calton Pu and I have developed tools and techniques for assessing operational impact [7]; and, we have also investigated ways to implement these techniques in a distributed manner [8]. We will now review related research in order to highlight our contributions in these areas.

3.1 Impact Analysis

Aguilera, et al. note that there are still a significant number of administrators who perform impact analysis manually, based on best practices and rules of thumb [9]. Unfortunately, manually analyzing the impact of a particular change does not scale well as the size of the enterprise system increases with respect to the number of devices, and the scope and complexity of the subcomponents and applications. There have been attempts to standardize and automate impact analysis to overcome these challenges.

In some cases, operational impact is determined in a relatively static fashion. As one example, the Department of Defense (DoD) has addressed the subject of operational impact as part of its Information Assurance and Computer Network Defense (IA/CND) program [10]. This document presents the three Mission Assurance Categories (MACs) that can be assigned to DoD information systems. MACs are assigned by the chief owners/operators of a system, and reflect the importance of the information they process

relative to the achievement of various missions: MAC I systems are considered vital to mission success, and the operational impact of losing such a system could very likely result in mission failure; likewise, the loss of a MAC III system would have minimal, if any, impact on the outcome of the mission. The MAC assignments could support mapping the loss of a specific application to the operational impact – however, there are difficulties with this approach.

First, the users do not always assign MACs to all relevant systems; and if assigned, the MACs are not always updated in a timely manner in response to changes in the business workflow and/or system configuration. MACs are normally assigned at system level, so it remains difficult to determine how the loss of a subcomponent will affect the overall operation of the system. Finally, if a system handles a range of information levels (from routine to critical), then the highest applicable MAC is assigned by default. This default labeling can be deceptive, especially if the system is used for critical data processing only in the most extreme (and infrequent) cases. In short, some systems with a high MAC rating receive an unnecessarily large amount of management and maintenance focus, while other mission critical systems are neglected or overlooked.

More dynamic methods for analyzing impact have also been developed. Hanemann et al. propose a high-level framework that focuses on the importance of impact analysis in ensuring that quality of service (QoS) metrics and Service Level Agreements (SLAs) are met [11]. They articulate the need for impact analysis to be integrated with service-oriented event correlation, in order to determine which services have been affected by a resource failure. Event correlation typically deals with resource-level events (e.g. the crash of a specific application server), and data on these events can

be collected from customer technical reports, provider service monitoring, etc. Jobst and Priessler make a similar argument using “event clouds” [12]. Business Activity Monitoring (BAM) is done at the level of key business objectives and metrics, and BAM tools are used to collect business activities in a higher-level event cloud. The intent is to employ user-defined use cases, and event correlations and patterns, to map events in the higher-level, business-oriented event cloud to events in the lower-level, technically oriented event cloud. Thereska, Narayanan and Ganger also address the need to consider impact analysis in a proactive manner [13]. More specifically, they propose a “what-if” approach that supports interactive exploration of the results of system changes, which include planning for deliberate configuration and tuning changes, as well as considering potentially unplanned resource failures. Singh, Koropolu and Voruganti examine the importance of impact analysis in the more specific scenario of file storage [14], while Hariri et al. focus on impact analysis as related to system and network security [15]. These approaches highlight at least two important aspects of impact analysis: determining the user-level applications, systems and services affected when a specific resource fails; and, expressing the resulting analysis in a clear, concise and business-relevant manner.

Jashki et. al. address the important issue of using impact analysis methods to reduce operational impacts when modifying software systems [16]. They propose a static technique maintains an alignment between the business processes of the organization, and the software systems used to support those processes. They then demonstrate the success of their technique when applied to different open source project repositories. Our current techniques simply monitor the system at runtime, and then use the most current dependency topology to assess future impacts. Similarly, Walker et. al. present a

technique for assessing the technical risk in an evolutionary development setting [17]. Their technique can be used vertically within an organization, allowing the development staff to discuss and assess the risks together with high-level management. Even with our current focus on dynamically-oriented data collection techniques, there is potential in adapting our methods to use some of these statically-oriented techniques in combination, which might improve the accuracy of our assessment results.

3.2 Dependency Discovery & Modeling

There has also been significant research on the importance of dependency analysis in determining the impact of a resource failure. Kar, Keller et al. address the problem of discovering and enumerating dependency relationships between applications and lower-level services in a networked environment, and recognize that this is a difficult problem having static and dynamic aspects [18]. They establish a multidimensional framework for classifying dependencies, and develop these concepts further in later research. To determine statically-based dependencies, they propose a repository-based approach, which discovers dependencies by analyzing the data commonly found in most operating systems – for example, the Object Data Manager/ODM in AIX, and the Registry in Windows. Their process then matches the data based on key fields in the repository structures. They later extend this approach to manage application services hosted by Network Service Providers [19].

Other research has focused on determining actively based dependencies. Kar, Keller et al. also propose an Active Dependency Discovery procedure that captures dynamic dependency information by perturbing the monitored system [20]. The

perturbation results are analyzed statistically to infer causal relationships. Ensel also proposes an automated dependency discovery method, but uses neural networks instead of the statistical methods normally advocated by other approaches [21]. This is an attempt to address the lack of direct dependency information and scalability issues commonly encountered with very large, heterogeneous networks. Chen, Kiciman et al. propose an application-generic methodology to better understand inter-component relationships and diagnose problems [22]. Their approach exploits a key observation that most dynamic, distributed systems have a single system-wide execution path for each request; consequently, their methodology traces runtime paths to collect the control flow, resources and performance characteristics associated with a request, and uses correlation analysis to determine system structure, and deduce resource failures. They discuss the development of the Pinpoint system, which uses instrumented J2EE middleware, and an application-layer packet sniffer, to trace client requests and detect both internal and end-to-end failures [23]. They then apply their Pinpoint system to detecting application failures in component-based internet services [24]. Pinpoint is first used to build a dynamic model of the system to establish a pattern of "normal" behavior. This model is then used to determine anomalous behavior, and the likely occurrence of a high-level fault.

In automating dependency discovery, the proposed approaches vary significantly in the amount system changes required. Kar, Keller et al. take a relatively passive approach, where application programming interfaces are used to pull data from existing operating system repositories, and generate static dependencies. In contrast, Chen, Kiciman et al. instrument the middleware system to capture runtime trace data, and

Hanemann et al. employ agents to identify critical resource points and calculate impact factors in real time. Kar and Keller's later research generates dynamic dependencies, but requires that the monitored system be instrumented and perturbed in an offline state. Some other approaches can be used with a variety of passive and active data gathering approaches. Though instrumentation, offline perturbation and other system modifications might yield better overall results, these techniques can also make it much more difficult to implement and manage dependency discovery in practice, especially in production environments.

3.3 Data Mining & Usage Monitoring

Expressing the results of the impact in a clear and understandable manner is very important. We firmly believe that to make the technical impact relevant to users, you must have some understanding of how the users utilize the technical resources available to them. Data mining has been used for system, operations and application management. Srivatstava et al. investigate approaches for mining usage data in web accesses [25], while Van der Aalst et al. survey approaches to mine business workflow patterns [26]. Jobst and Priessler also address presentation concerns, and propose enterprise performance cockpits and dashboard layouts that depict the enterprise system's performance [12]. We believe these mining techniques can be adapted to end user access patterns, to support the presentation of business-relevant impact results. Our approach goes beyond previous work by abstracting and integrating system level events and application level events. As an example, in the mining of usage data to detect business workflow patterns, Aalst et al. mention the exploitation of timing data as an open

problem in workflow mining. We use timing as the underlying fabric on which we integrate events from all system and application components.

3.4 Commercial Solutions

There are commercially available solutions that attempt to address the problem of assessing operational impact. One such example is the SMARTS Business Impact Manager system produced by EMC^2 [27]. Numerous major Internet Service Providers use the SMARTS system to provide root cause analysis and business impact assessment functionality. SMARTS uses mathematical techniques to model the system environment, and also updates the model automatically when changes are detected. This provides some advantages over systems that require dependency rules to be manually generated and maintained, especially in large, dynamic environments. The Business Impact Manager functions by assigning weights to various elements of the system environment, and then using the numbers and weights of affected elements to calculate values for business impacts. Unfortunately, the business impact assessment functionality still requires manual assignment and adjustments of weights.

There are other commercial systems that also address key aspects of assessing operational impact. The IBM Tivoli Application Dependency Discovery Manager can be used to automatically discover dependencies between various applications, systems and infrastructure components [28]. It can also be used to poll the system over time, to detect configuration changes. It operates as an agent-less system: the Discovery Manager software can be installed on a server of adequate computing power, and configured to contact each workstation or server to be monitored over the local network. This setup still requires that each computer to be monitored be configured to permit remote access,

and that commands be enabled on the machine to permit complete visibility of open files. Also, internal servers (e.g. web, database, application) and other infrastructure components (i.e. network routers) must also be specifically configured to allow the Discovery Manager to access them and collect the information needed to build the dependency model. While this is usually very feasible for reasonably fixed, static environments with sufficient bandwidth connectivity, this could be more challenging to implement on more dynamic, lower-bandwidth connectivity systems. The requirement to allow the Discovery Manager to have root access to such a large number of components and to conduct numerous network and port scans while collecting dependency information could also raise security concerns in some environments.

3.5 Research Most Similar to Our Approach

3.5.1 Operational Impact Analysis

The framework proposed by Hanemann automatically determines the impact of resource failures with respect to services and Service Level Agreements (SLAs) [11]. This requires a holistic view of the service provisioning structure, including knowledge of the dependencies between the offered services, subservices and resources, and the customer's SLAs. Their framework is applicable in two time perspectives: in the short-term, where current failures are evaluated to determine the services impacted, especially those services covered by one or more SLAs; and mid-term, allowing the service provider to simulate the effects of resource failures for planning purposes. One of the strengths of their framework is that it covers a significant portion of the impact analysis problem, including the modeling, dependency analysis, and SLA monitoring and management factors. This is very rare: most approaches focus solely on one of these areas.

Our proposal addresses dependency discovery, while their framework assumes that the dependencies are generated by other systems. Our proposal also varies in our method of assessing impact. We submit that SLAs are normally written to ensure a certain level of service for the most operationally critical resources (e.g. programs, services), and that SLAs are fairly static once established. Users, however, will normally use whichever resources are available in order to complete their assigned business-level tasks in the most efficient way possible. In some cases, users are restricted to the resources designated in the SLA; in other cases, the SLA should be reviewed periodically to ensure that the designated resources are still important from an operational perspective. Our approach assesses impacts in terms of the user-level resources commonly accessed during the time frame being analyzed. If users change the resources that they use to complete business-level tasks, our approach will detect these changes; and, since the usage patterns are updated over time, our impact assessments will remain synchronized with the operationally relevant resources. In this manner, our approach also supports integrating impact analysis with service-oriented event correlation, such that resource failures can be utilized as input for impact analysis.

In a supporting vein, Stanley, Mills et. al. correlate network services with operational mission impact [29]. The object is to align IT services with the supported mission services, which they do using their Mission Service Automation Architecture (MSAA). Their approach is based on a framework provided by the Information Technology Infrastructure Library (ITIL), and requires that the IT providers and end users identify the initial linkages between IT services and supported mission services. This requires that IT providers have a solid understanding of the mission/business nature

of their organization, and also that the end-users have a solid understanding of how they make use of the IT services to accomplish their objectives. Our contention is that this "shared knowledge" amongst IT providers and end users is not as common as is normally desired; and, because of this fact, automated approaches are needed to "jump start" this communication between the two parties. That is why we employ automated dependency discovery techniques in our approach.

3.5.2 Black-Box Monitoring

Given our goal of developing an approach that minimizes installation and management overhead, we also examined research that proposed more lightweight, black box approaches to monitoring. Aguilera et al. model a distributed system as a graph of communicating nodes, and obtain message-level traces of system activity as passively as possible and without any knowledge of node internals or message semantics [9]. This approach requires no modifications to applications, middleware or messages. Similarly, Mahajan et al. present an architecture for user-level Internet path diagnosis that requires minimal special privileges or network support [30]. These efforts lead me to believe that a black-box monitoring approach is feasible, and worth further study.

3.5.3 Dependency Discovery

While Keller et al. use static information from operating system repositories, we sought to use active information, with the intent of capturing runtime dependencies and user access patterns. Sitaraman et. al. [31] and King et. al. [32] employ the Backtracker tool, which is used to help system administrators identify potential entry points for intrusion detection. Backtracker logs runtime events that can then be used to infer dependencies between operating system objects. Similarly, other researchers have demonstrated

different ways to leverage the traceroute command to map various environments, including client-server and peer-to-peer networks [33], [34], [35]. This information can be collected using unmodified operating system commands in most cases, supporting our lightweight, black box approach.

The Backtracker system induces dependency relationships between objects by tracking events in which one object affects the state of another object. They denote a dependency to a source object from a sink object as source → sink, along with a time interval to reduce false dependencies. For example, a user logging into a computer using a certain password file establishes a file → process dependency because the login process needs data from the file. They are focused on three types of dependencies based on the objects being monitored: process/process, process/file and process/filename. A logger component is used to collect event information, and the logger can be implemented as part of the Linux kernel, or as a Linux loadable kernel module.

Our process induces similar types of dependencies using very similar reasoning. One distinction is that we collect event information by issuing operating systems commands instead of instrumenting, or loading modules into, the Linux kernel. This supports our goal of minimizing application, middleware or system modifications in order to make our system more likely to be used. Another major distinction in our approach is that their model focuses on processes, files and filenames; our model includes other objects such as users, devices, network ports, remotes sites and routers. Another minor distinction is that we do not use individual time intervals for each of the dependencies; however, we only induce dependency relationships for event information collected at the same time (i.e. within the same snapshot).

3.5.4 Scalability

Another goal is that the approach be scalable to large systems. Mortier, Isaacs and Barham developed the Anemone system, which uses end systems to perform network management [36]. The user workstations are instrumented to act as 'traffic sensors' and collect flow data in a distributed manner. This flow data is combined with topology data collected from the routing protocols to provide a richer network management dataset. This approach takes advantage of the idle cycles, disk space and network bandwidth available on the individual workstations, as well as the fact that placing the data sensor closer to the end system offers significant advantages in being able to examine the original unencrypted, non-partitioned/"packetized" traffic. Some of the same advantages are also valid for our focus on interactions with applications, files, and other user-accessible resources.

Anemone treats end-systems as 'traffic sensors' and combines flow-data from these systems with topology data inferred from routing protocols. Our approach does treat end-systems as 'data sensors', with a broadened focus on areas such users, programs and files, as opposed to Anemone's focus on network management. Also, our approach uses traceroute data collected from the end-systems to infer network topology, as opposed to Anemone's approach of gathering topology data by monitoring the Link State Advertisements (LSAs) that are normally flooded to the routers.

CHAPTER 4

PROCESSING DATAFLOW & OVERVIEW

In this section, we will describe our proposed framework for assessing operational impact. Our approach is motivated by the challenges addressed earlier, and the difficulties faced by current approaches. First, we will focus on the dataflow and algorithms that are used for the dependency modeling and user access monitoring aspects of assessing operational impact. We will discuss the basic architecture of our approach, along with specific implementation details, in a later section.

4.1 Impact Assessment Dataflow

Our approach is divided into four basic phases: Collection, Discovery, Analysis and Mining. Figure 4 captures the basic dataflow and sequencing of the phases.

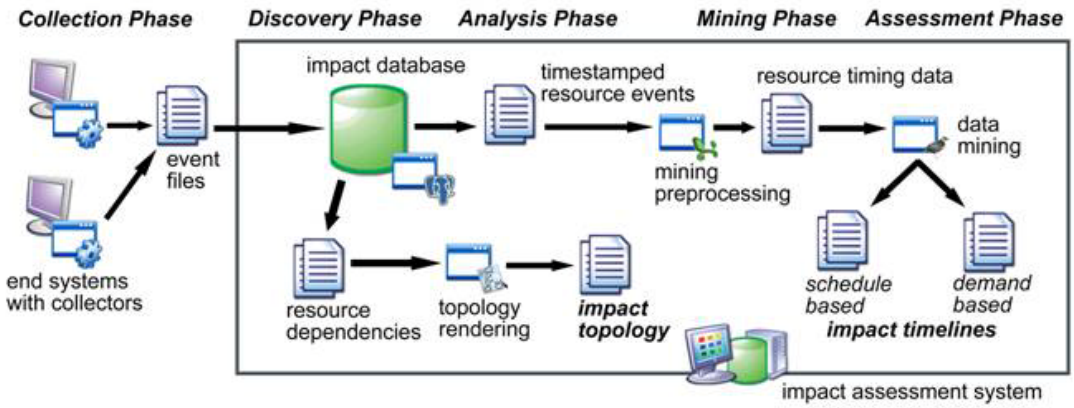

Figure 4 - Early Version of the Dataflow Architecture

During the ***Collection Phase***, common operating system commands are used to extract information from end-user workstations. This information is collected on a relatively

frequent basis, every few seconds or minutes, and the data is stored in files for use during the Discovery phase. Command options are selected to standardize the data output, and minimize the use of system resources. The raw data is processed to eliminate non-essential attributes from the command output, and handle other syntax issues. The data is also time-stamped to ensure that the data elements for each specific time period can be linked correctly during the Discovery phase. The time-stamping also supports user access monitoring during the Mining phase.

During the ***Discovery Phase***, the raw data files are used to construct a dependency model for various system components. This dependency model can be used to compute transitive dependencies between components and applications, allowing Administrators to more clearly and concretely explain these relationships as part of the impact assessment process. The Discovery phase also includes determining whether components are local or global. Local components are only relevant impact-wise to the workstation where the data was collected, while global components (i.e. routers) may have an impact on multiple workstations within the monitored system. The local/global designation supports a system-wide, single-search method for assessing the impact of a designated component, as opposed to requiring a separate search on each workstation. The dependency model is further examined as part of the Analysis phase.

The ***Analysis Phase*** employs a top-down search methodology using the dependency model to identify those system components that affect one or more users. This reduces effective size of dependency model, making searches to determine if an impact occurred quicker. The Analysis phase also calculates the occurrence frequency for each of the dependencies to identify the best candidates for the mining phase. The

basic idea is that dependencies that do not occur often enough will not generate enough data to satisfy the minimum support and confidence thresholds during the Mining phase; therefore, eliminating them now reduces the overall amount of data to be processed, and improves performance. Similarly, if a certain dependency appears during almost every collection period, then that dependency is basically continuously active, and attempting to detect any other usage patterns will likely not generate any significant information. Consequently, the Analysis phase produces a user-focused set of dependencies that are most likely to yield significant usage pattern information during the Mining phase.

Finally, the ***Mining Phase*** uses the user-focused dependency information to detect usage patterns for the system components. First, the information is translated into a format more suitable for data mining, with a focus on the user-level applications. Then, the data is mined for scheduled and on-demand timing patterns, as discussed earlier in the paper. Mining for scheduled patterns involves using decision trees, association rules and other common data mining tools to determine if an application will be active at a certain day, date, month and/or hour of the day with a certain level of confidence. Mining for on-demand patterns involves using autocorrelation analysis to determine if an application will be active within a certain time window from t_0 to $(t_0 + \Delta t)$ given the set of applications that are active at time t_0. Administrators can use the mining results to better quantify the probabilities that operational impacts will occur after a specific technical event has occurred, or to predict potential impacts for planning purposes – for example, when determining the best period to apply critical security patches.

The Discovery and Mining phases capture the dependency modeling and user access monitoring aspects, respectively, that are essential to assessing operational impact

effectively. The Collection and Analysis phases are essential to transforming the data into standardized formats suitable for further processing, and organizing and reducing the amount of data to improve the efficiency of the overall process. We will cover each of these phases in more detail in the following sections.

4.2 Collection Phase

The Collection Phase leverages common operating system commands to accumulate data about the current state of the workstation being monitored. My current research focuses on Linux and Unix based operating systems. We use cron-activated batch files to capture data about the current state of the workstation being monitored. The batch files execute common Linux operating systems commands like *w*(), *ps*(), *lsof*(), *df*() and *traceroute*() to collect data about users, programs and processes, open files, and remote sites. The command schema relationships are shown in Figure 5. The batch files also format the output for further processing during the Discovery Phase. The state of the workstation includes information about which users are logged on, which processes are currently running, and which files are open, among others. It also uses some other commands for time-stamping and formatting the data.

The ***w( )*** command is used to determine who is logged onto a system, and what each user is doing. More specifically, we use the command "w –s", where the "–s" option is used to display information in a summarized format. Using the "–s" option omits certain fields, like those used to measure the time consumed by the current foreground and background processes.

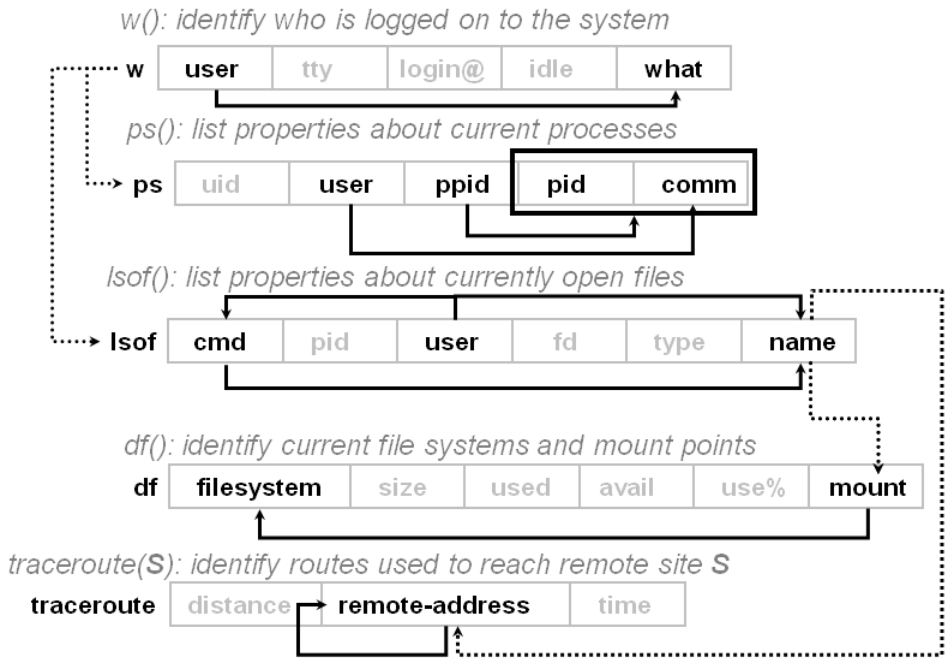

Figure 5 - Operating System Command Schema & Key Field Relationships

A sample of the data output from the w() command is given here:

```
poseidon% w -s
 13:05:52  up 71 days, 20:09,  5 users, load average: 0.00,
0.02, 0.00
USER        TTY   FROM              IDLE  WHAT
adams       pts/0 achilles.cc.gt.a  4days -bash
brown       pts/1 lawn-128-61-114-  0.00s -bash
chelsea     pts/5 c-24-30-25-54.hs  0.00s w -s
brown       pts/4 lawn-128-61-114-  1:19  less INSTALL
```

The USER field represents the actual login name for user. The names displayed here, and elsewhere in the paper, are not the actual user names – they have been changed to respect the privacy of the real users. This substitution does not affect the accuracy of the results. If actually implemented in a production environment, a properly authorized administrator would use the real login names in order to more accurately assess operational impacts.

The TTY field represents the name of the terminal the user is accessing, while the FROM field displays the host from which the user is logged in. As shown above, the command has automatically resolved the network addresses into names. Ideally, we would also employ the "–n" option to avoid this translation, if available. The network names are frequently truncated, which makes it much more difficult (though not impossible) to use them later when linking records during the Discovery phase; consequently, using network addresses is preferable. Also, eliminating the address-to-name translation would consume less time and computing resources. Unfortunately, though this option is available on some operating systems (i.e. BSD/Mac OS X), it is not available on all Linux systems.

The IDLE field displays the time since the user last entered any input, while the WHAT field displays the current command and options. For the Discovery phase, the only data required is contained in the USER, FROM and WHAT fields. We used grep() and other commands to eliminate unneeded data, and to format the output into the following schema:

w-data := (user-name, access-site, access-program)

where "user-name" represents the login-name, "access-site" represents remote network names used to access this terminal, and "access-program" represents the name of the command being executed to support remote access. The data from the w() command is stored in this format in a file, and loaded into a database during the Discovery phase. The records are also tagged with time-stamp information, and the name of the workstation on which the command was executed. I've omitted the time-stamping information and

workstation name from this and the following schema definitions for clarity, since this information is relevant only during the Discovery and Mining phases.

The ***ps( )*** command is used to display a list of active processes. Normally, the ps() command lists all processes with the same effective user ID as the current user, and associated with same terminal as the invoker. More specifically, We use the command "ps -eo user,pid,ppid,comm", where the "–e" option is used to designate all processes, and the "–o" option allows me to specifically designate the columns to be displayed. A portion of the output from the ps() command is given here:

```
poseidon% ps -eo user,pid,ppid,comm
USER        PID         PPID        COMMAND
root        1           0           init
root        1212        1           sshd
xfs         1406        1           xfs
daemon      1424        1           atd
root        20506       1212        sshd
smith       20508       20506       sshd
...
```

The USER field is defined as in the w() command above. It is important to note that the USER field here contains the names of real users (e.g., philip, smith) as well as the names of special system accounts (e.g. root) and accounts used to manage services and other long-running processes (e.g. xfs, daemon). The PID and PPID fields contain the process and parent process identifiers, respectively. These values can be used to establish which parent process issued a fork() command to create a child process, thus establishing a "process tree." As an example, the process with pid 1212 spawned the process with pid 20506, which later spawned the process with pid 20508, which is owned by smith and currently running the sshd() program.

The COMMAND field lists the actual name of the program that is currently running inside the process space. The program running within a process can be changed

using the exec() command. Since process identifiers can change between different invocations of the ps() command, we only use the PID and PPID fields to establish relationships between the data in the USER and COMMAND fields. Monitoring the ps() command allows us to monitor the situation where different programs are executed in the same process space, which is not necessarily possible with other forms of monitoring. For example, many Linux and Unix systems offer support for process accounting, which is normally managed using the ***accton( )*** command [37]. When enabled, the kernel writes an accounting record each time a process terminates, where the record contains the user ID, controlling terminal, and program being executed (along with other information) at the time the process was terminated. This system does not record the names of programs that executed in that process space prior to termination. For example, if a process is started using program A, which then executes program B, followed by execution of program C, only program C will be recorded in the accounting record for that specific process. While using accton() ensures that we will capture a record for every process that terminates, along with the terminating program, executing ps() on a frequent basis allows us to also capture the initial and intermediate programs that execute in each process space as well.

The data for all of the given output fields is needed for the Discovery phase. We used grep() and other commands to format the output into the following schema:

ps-data := (user-name, process-id, parent-id, program)

where “user-name” represents the login-name; “process-id” and “parent-id” represent the process identifier information; and “program” represents the name of the command being

executed in the process space at this moment in time. The data from the ps() command is stored in this format in a file, and loaded into a database during the Discovery phase.

The ***df( )*** command stands for "disk free", and displays the names and space statistics for the accessible file systems. By default, it displays statistics (e.g. total space, available space, percentage of space used) only for those systems for which the user has read access. The command also displays the file system roots, which represent the directories below which the file system hierarchies appear. We use the df() command to link files to these directories, and then to link the directories to the drives on which they are located. The file system's name is contained in the Filesystem field, and the location of the directory hierarchy is contained in the Mounted on field. Some sample output from the df() command is given here (the "Mounted on" column is boldfaced for clarity):

```
moss-pinata:~ sylviamoss$ df -ah
Filesystem                          Size   Used  Avail Capacity
Mounted on
/dev/disk0s2                       186Gi  136Gi   50Gi    74%
/
devfs                              106Ki  106Ki    0Bi   100%
/dev
fdesc                              1.0Ki  1.0Ki    0Bi   100%
/dev
map -hosts                           0Bi    0Bi    0Bi   100%
/net
map auto_home                        0Bi    0Bi    0Bi   100%
/home
//GUEST:@simpleshare:139/NetFolder  149Gi   59Gi   89Gi    40%
/Volumes/NetFolder
http://idisk.mac.com/markmoss/       10Gi  183Ki   10Gi     1%
/Volumes/markmoss
```

The Filesystem and Mounted on commands are defined above. We do not make use of the other statistics at this time, but they could be incorporated in future versions of the system. For example, most of our impact assessments are focused on component failures; however, significantly degrading the performance of a component can also cause operational impacts. Consider the case of a disk drive that is almost full; in many cases,

the lack of free space on the drive can cause intermittent failures and faults that are otherwise very difficult to determine. We could use the Capacity field to detect drives that are overloaded (e.g. 98+% full), and then assess the operational impact on the users, files and programs that access those corresponding file systems. The data for all of the given output fields is needed for the Discovery phase. We used grep() and other commands to format the output into the following schema:

df-data := (file-system, mount-point)

where "file-system" represents the name of the file system, and ; "mount-point" represents the directory location. When linking files to their directories, we attempt to match the file with the most specific mount point that is available. The file names that are gathered from the lsof() command are normally the fully-qualified file names, which include the complete directory path from the root directory. Suppose that we are using the df() data from our example above, and we have just received the file **/Volumes/fizz/myfile.txt**. Our first attempt would be to match some valid prefix of /Volumes/fizz/myfile.txt against one of the known mount points. **/Volumes/fizz** cannot be unified with either **/Volumes/Netfolder** or **/Volumes/markmoss**. If the simple **/Volumes** mount point existed, we would use its' corresponding file system. However, since it doesn't, we continue our matching attempts until we reach the root directory (/) mount point, which matches all directories by default. Consequently, we would link the file Volumes/fizz/myfile.txt to the root directory / file system, which is linked in turn to the device */dev/disk0s2*. The data from the df() command is stored in this format in a file, and loaded into a database during the Discovery phase.

The ***lsof( )*** command, by default, displays a list of all open files corresponding to every process currently running on that computer. Linux and Unix use file structures for many activities, and an open file can represent a regular file, a library, a directory, a stream, or a network socket; consequently, the output from a default lsof() command is normally very large. We used command line-options to better divide the resulting output into two sets of data: file-oriented and network-oriented data. Also, since certain options of lsof() can be resource intensive, We selected the options carefully to minimize the impact on the user's operations. One specific example is that we used the "-n" and "-P" options to prevent translation of network addresses and ports, respectively, into names. This translation requires the system to perform Domain Name Service (DNS) lookups, along with other unnecessary and time-consuming operations. Also, the network names are often truncated during output formatting, which makes record matching during the later phases more problematic. A sample of the output from the file-oriented version of the lsof() command is given here (the "NAME" column is boldfaced for clarity):

```
poseidon% lsof -nP
COMMAND     PID     USER   FD   TYPE  DEVICE     SIZE    NODE
NAME
init          1     root  mem    REG     8,1    27036    1815225
/sbin/init
dhclient    958     root  txt    REG     8,1   344544    1815307
/sbin/dhclient
syslogd    1002     root  txt    REG     8,1    33861    1816385
/sbin/syslogd
klogd      1006     root  txt    REG     8,1    27080    1816384
/sbin/klogd
portmap    1018      rpc  txt    REG     8,1    12476    1815448
/sbin/portmap
rpc.statd  1037  rpcuser  txt    REG     8,1    30808    1815449
/sbin/rpc.statd
ypbind     1112     root  txt    REG     8,1    30816    1815471
/sbin/ypbind
sshd      27793     root  mem    CHR     1,5              40233
/dev/zero
```

```
csh       27796      sam  rtd    DIR     8,1     4096          2
/
lsof      27830      sam   2u    CHR   136,1                   3
/dev/pts/1
```

The COMMAND, PID and USER fields are as defined for the ps() command above. The FD and TYPE fields are the File Descriptor and Node Type, respectively. These attributes are used to identify the different kinds of files: for example, an FD value of "txt" represents a text file containing program code or data, which is normally has a TYPE value of "REG", which stands for a regular file. Since the focus is on user-level programs and data, we can filter out the appropriate records by piping the output through the appropriate grep() commands. The DEVICE field can be used identify where the file is stored, is listed in a <major number>, <minor-number> format. The NAME field represents the actual name of the file. For the Discovery phase, the only data required is contained in the COMMAND, PID, USER, DEVICE and NAME fields – the SIZE and NODE fields are not required. We used grep() and other commands to eliminate unneeded data, and to format the output into the following schema:

lsof-file-data := (program, process-id, user-name, device-name, file-name)

where "program", "process-id" and "user-name" are defined as above; "device-name" represents the identity of the storage device for the file; and "file-name" represents the name of the file. The data from the file-oriented version of the lsof() command is stored in this format in a file, and loaded into a database during the Discovery phase. The "–i" command-line option can be used to generate the network-oriented version of lsof() by limiting the output to IPv4 and IPv6 records. A subset of the output from the network-

oriented version of the lsof() command is given here (the "NAME" column is boldfaced for clarity):

```
poseidon% lsof -nP -i
COMMAND     PID     USER   FD   TYPE DEVICE SIZE NODE
NAME
dhclient    958     root    5u  IPv4   1124       UDP
*:68
rpc.statd  1037  rpcuser    7u  IPv4   1311       TCP
*:32768 (LISTEN)
ypbind     1112     root    5u  IPv4   1442       TCP
*:869 (LISTEN)
cupsd      6513     root    2u  IPv4  16610       UDP
*:631
sshd      20843     sam    4u  IPv4  75085       TCP
130.207.5.228:22->24.30.25.54:50267 (ESTABLISHED)
```

The NODE and NAME fields contain different information when referring to network records. The NODE field normally contains the unique i–node address when dealing with file-based information; here, it contains the transport-level communications protocol. Here, the NAME field contains the network information instead of the file name. The network information is displayed in the $\langle network - address \rangle$: $\langle network - port \rangle$ format for open sockets. The information for established connections is displayed in the format:

$\langle local - address \rangle : \langle local - port \rangle \rightarrow \langle foreign - address \rangle : \langle foreign - port \rangle$

In this case, we use awk() and other commands to extract the address and port data, and to format the output into the following schema:

lsof-network-data := (program, process-id, user-name, local-address, local-port, foreign-address, foreign-port)

where the foreign-address and foreign-port fields have values for records with established connections, and are null otherwise. Similar information is extracted from the ***netstat( )*** command, using the "–a" and "–n" options. The "–a" option requests all open sockets

and connection, and the "–n" option prevents address-to-name translation, similar to the "–n" and "–P" options for the lsof() command. A sample of the output from the netstat() command is given here (the "State" column is boldfaced for clarity):

```
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address
State
tcp        0      0 0.0.0.0:512             0.0.0.0:*
LISTEN
tcp        0      0 130.207.5.228:697       130.207.117.14:111
TIME_WAIT
tcp        0     48 130.207.5.228:22        24.30.25.54:55001
ESTABLISHED
udp        0      0 0.0.0.0:514             0.0.0.0:*
```

where the "Local Address" and "Foreign Address" fields are defined and formatted as in the lsof() command. We use awk() and other commands to format the data into the following schema:

netstat-data := (local-address, local-port, foreign-address, foreign-port)

The data from the lsof() and netstat() commands are stored in these formats in distinct files, and loaded into a database during the Discovery phase.

The ***traceroute( )*** command is used display the route that packets take to reach a designated network host. Because it uses multiple ping() requests, executing traceroute() commands can create a significant load on the network if used without caution. We use the "–n" and "–q" command line options to minimize the load on the network. The "–n" option avoids the address-to-name lookup for each hop along the path, as in the lsof() and netstat() commands. Also, traceroute() normally send three ping() requests to each gateway along the path, in order to measure the average round-trip time to that gateway.

Since we are not interested in time measurements, we use the "–q" option to issue only one ping() request per gateway, in order to learn the gateway's network address.

Also, the traceroute() command requires an extra, dynamic argument that was not required in the previous commands: a target network address (or name) must be designated. The target network addresses used will be drawn from the foreign addresses collected from the lsof() and netstat() commands. A portion of output from the traceroute() command is given here:

```
poseidon% traceroute -n -q 1 24.30.25.54
traceroute to 24.30.25.54 (24.30.25.54), 30 hops max, 38
byte packets
 1  130.207.5.1  0.388 ms
 2  130.207.251.1  0.396 ms
...
11  *
12  *
...
```

The gateway addresses are listed in increasing number of hops away from the source address from which the traceroute() is being conducted. Certain gateways will not return their address, and so the traceroute() result will return a "*" for that gateway. We use grep() and other commands to remove these records from the resulting output, to re-sequence the records after removing those records where the gateway didn't return an address, and to format the output into the following schema:

traceroute-data := (target-address, sequence-number, gateway-address)

where "target-address" is the network address of the host the traceroute() was trying to reach; the "sequence-number" corresponds the order of "gateway-address", from the source address to the target address. The data from the traceroute() command is stored in this format in a file, and loaded into a database during the Discovery phase.

There are other systems and approaches that can be used to generate more comprehensive coverage of the system dependencies; for example, Unix, Linux, Mac OS, Windows, and a number of other operating systems offer different built-in logging facilities. While we continually look for a way to leverage these built-in facilities, we also have to ensure that they will provide enough information to allow us to link resources during the Discovery Phase. My experience is that some of these systems record when specific components are used, as opposed to recording when the dependencies between components are active. Time stamping the components, as opposed to the dependencies, can cause ambiguity problems when assessing impacts. As an example, consider the problem shown in Figure 6.

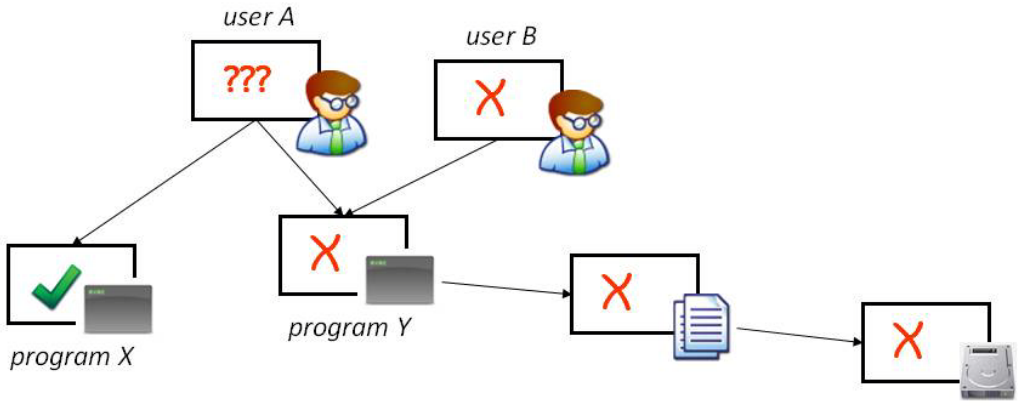

Figure 6 - Ambiguity Problem (Insufficient Log Information)

The hard drive component has just failed, preventing program Y from being able to access the file on the hard drive. User B, who only uses program Y, will be operationally impacted if he/she is using program Y during the failure duration. Consider user A, where he/she might be using program X or Y. Program X is operating normally, and users access program X will not be operationally impacted. Suppose that our logs only indicate events at the component level: for example, programs X and Y are active, and

user A is online. If the data is time stamped at the component level, as opposed to the dependency level, then we will not be able to determine whether user A is being impacted by the failed component. A key aspect of the data collected during the Collection Phase is that it contains enough information to determine which programs are being executed by which users, which files are being accessed by which programs, etc. If we could leverage more built-in logging systems, and ensure that the data received would be sufficient to generate the required dependencies, then we could improve the overall coverage of the systems dependency topology, consequently improve the quality of the assessments.

4.3 Discovery Phase

4.3.1 Dependency Model – Resources and Zones

The Discovery Phase uses the collected data during the Collection phase to establish the dependency relationships between various components. The topology is modeled as a directed graph, where the nodes of the graph represent system ***resources***, and the edges represent ***dependency relationships***. An edge from resource A → resource B means that resource A is dependent on resource B in some defined manner. My proposed model identifies seven distinct resources, and eleven dependency relationships between those resources. Figure 7 displays the dependency model, including the resources, dependencies, and zones.

The resources are represented as 3-tuples of the form <***zone, type, identifier***>, where ***identifier*** is the distinguishing name within each resource type. The ***identifier, type*** combinations are used to avoid machine-wide naming conflicts, since two different resources may have the same identifier; for example, a user's login name may be identical to the name of an executable program. The nine different ***types*** of resources are:

- *USERS* – represent actual users, who may access the system at a local terminal, or from a remote terminal via some communication program like ssh(); can also represent "system accounts", where the name represents an account created to manage one or more system services
- *PROGRAMS* – represent the executable code segments that are used to access, manage and modify data
- *SITES* – represent remote locations, such as web sites, that contain data and/or provide computing services
- *FILES* – represent collections of data in various formats
- *DEVICES* – represent the system elements that store files and other data; they can be local elements like hard drives, or network accessible like storage appliances

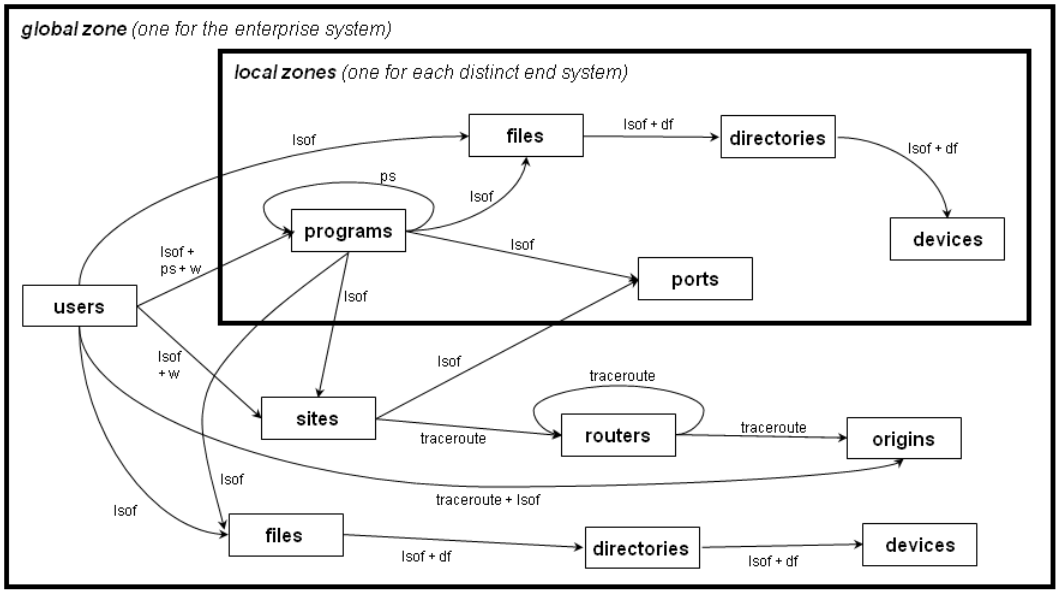

Figure 7 - Dependency Topology Model

- *PORTS* – represent the transport-layer elements used to coordinate and de-conflict communication with various local and remote services and sites
- *ROUTERS* – represent the elements used to forward data between different sites; for our purposes, the term "router" is used to represent a generic forwarding system: no distinction is made between routers, switches, and any other packet forwarding systems
- *DIRECTORIES* – represent the hierarchies used to organize and help manage access to files
- *ORIGINS* – represent the computers and systems used to communicate with remote sites via routers

Resources can be local to a specific machine, or accessible by two or more machines within the system being monitored. ***Zones*** are used to capture this distinction, and can be either local or global. If a resource is accessible on the local computer only, then the machine-name, or some other uniquely identifying tag, is used for the zone value. If a resource is network accessible by two or more systems, then the static tag "*global*" is used for the zone value.

Zones support more comprehensive searching of the dependency topology while avoiding name conflicts. As an example, program X might be executing on two different computers – computer A and computer B. Assume that program X requires a library file F, which is stored on the local hard drive; and, a configuration file N, which is stored on a network accessible drive, which is used by all instantiations of X on the system. Since file F is stored locally, corruption of file F on computer A should only affect the execution of program X on computer A – program X's execution on computer B should

be unaffected. Likewise, if configuration file N is corrupted, it will potentially affect program X's execution on both computers A and B, along with any other programs that require N via remote access. A system topology view consists of the combined dependencies from one specific collection period. We use Graphviz to render the system topology and impact topology graphs [38].

In most traditional computer architectures, programs are loaded into memory on the designated computer, and then executed. Based on this, programs are always considered to be in the local zone. In most cases program files, along with some supporting library, configuration and data files, are stored on a local hard drive. In this situation the files, and the devices on which they are stored, are located in the local zone. However, files could also be stored on a network attached storage appliance, remote web site, or similar remotely accessible device. In these cases, the devices, and consequently the files stored on these devices, are located in the global zone. Similarly, remote sites, and the routers used to support communications between the local computer and these sites, are considered to be in the global zone. The ports used for communication are located in the local zone. This is consistent with current capabilities such as host-based firewalls, which means that port settings can be unique to, and reconfigured at, the local computer level. Finally, users are always considered to be in the global zone, since most modern systems support user login at many different computers.

4.3.2 Discovering Dependencies from the Event Data

In the following sections, we will describe how the data gathered during the Collection phase is used to identify and generate the dependency topology. Each data file, or combination of data files, is used to generate dependencies that are captured in the

snapshot set. Only data files that were collected at the same time (having the same timestamp) are used to generate a specific snapshot, to ensure the consistency of information like process IDs.

The w-data file contains data in the (user-name, foreign-address, program) format. Each record represents a specific user executing a program on the given computer. In many cases, the program is a shell to support user interaction, or remote access. If remote access is being used, then the foreign address will list the domain name or IP address of the remote terminal. This information can be used to generate the following dependencies:

- the user is executing the program to accomplish a goal, such that user-name → program
- if remote access is being used, then the user needs the remote site at foreign-address in order to access this system, such that user-name → foreign-address; also,
- the gaining access from the remote site depends on the successful execution of the remote access program, such that foreign-address → program

The ps-data file contains data in the (user-name, process-id, parent-id, program) format. Each record represents a specific user executing a program on a given computer. In some cases, the user-name does not correspond to a real user, but to a system account used to manage one or more services. The Analysis phase takes steps to identify and focus on the real users. Also, the programs execute in the context of a process space, which can be uniquely identified by the process-id. And since new processes are created by executing the fork() command on an existing process, the parent-id is the unique identifier of the

process used to create this “child” process. This information can be used to generate the following new dependencies in addition to the dependencies mentioned above:

- the program executing in the parent process has spawned a new process and program in order to accomplish one or more useful tasks, so that the parent program is dependent on the child program to accomplish its’ goals; consequently, parent-program → child-program

Since the name of the parent program is not given in the record, this information must be referenced from the appropriate record in the ps-data file.

The lsof-file-data file contains data in the (program, process-id, user-name, device-name, file-name) format. Each record represents a specific user employing a program (running in the process-id space) to access a file located on a specific device. As mentioned in the zoning discussion, the files and devices could be local to the computer where the data was collected, or network-based. Users must normally use one or more programs to access and modify the data in a file, and the programs used are often determined by the format and location of the file. If the program used to access a particular type of file is non-functional, then it will impact the user’s ability to manage that data. This information can be used to generate the following new dependencies in addition to the dependencies mentioned above:

- the user is also accessing the file to accomplish a goal, such that user-name → file-name
- the file can’t be read directly by the user, but must be accessed using one or more programs, such that file-name → program

- if the device that stores the given file encounters faults, then the user's ability to access that file will potentially be affected; consequently, file-name → device-name

The lsof-network-data file contains data in the (program, process-id, user-name, local-address, local-port, foreign-address, foreign-port) format. Each record represents a specific user employing a program to communicate via a local address and port. In some cases, the communication represents a program or service listening for activity; in other cases, communications have been actively established with a foreign address and port. Similar to the reasoning given in the lsof-file-data section, users must normally use one or more programs to access and modify the data located at a remote site. This information can be used to generate the following new dependencies in addition to the dependencies mentioned above:

- the program needs the local-port to be open and accessible to traffic for successful operation, such that program → local-port

Also, if the foreign address and port are valid (i.e. a connection is established), then the following dependencies can be generated as well:

- the user is also accessing the remote site to accomplish a goal, such that user-name → foreign-address
- the remote site can't be accessed directly by the user, but must be accessed using one or more programs, such that foreign-address → program
- the program and foreign sites require the local and foreign ports to be open to ensure successful communications; consequently, program → foreign-port, foreign-address → local-port and foreign-address → foreign-port

It makes sense to consider the netstat-data file in conjunction with the lsof-network-data file. Since the netstat-data file contains data in the (local-address, local-port, foreign-address, foreign-port) format, the lack of user-name and program data prevents us from generating certain dependencies directly. We can use the common local address and local port information from the lsof-network-data and netstat-data files in combination to generate more dependencies of the forms listed above.

The traceroute-data file contains data in the (target-address, sequence-number, gateway-address) format. Each record represents a generic routing/forwarding system located at gateway-address, located sequence-number of hops away from the local computer, used to communicate with the remote site located at target-address. Communication with the remote site normally depends on these routers being functional, and this information can be used to generate the following dependencies:

- communication from the local machine must make at least the first hop towards the remote site successfully; consequently, target-address → gateway-address when sequence-number =1
- in other cases, the router at sequence-number = k must forward it's data to the router at sequence-number = (k+1) such that $router_k \rightarrow router_{k+1}$

The main goal for the Discovery phase is to generate likely dependencies that can be derived from the collected data. We understand that dependencies generated might not capture all system dependencies; and, as such, we will need to be careful when assessing the impact on the user based on the failure of one or more resources in this model. As an example, consider communications with a remote site, and packets being forwarded along a path of routers. If one router fails in a well-designed network, chances are that

packets will automatically be rerouted along a different path, thus minimizing the impact to the users. The current dependency model will signal an alert that the users might be impacted, but different techniques can be used to confirm the actual impact, and to reconcile this data with the current model to improve it for future predictions. One method involves updating the dependency model when a fault is reported. In the router example cited above, a new traceroute() command to the target address would probably uncover the rerouted path, and this data could then be integrated into an updated dependency model. The dependency model provides a reasonable, first-order assessment of which users might be affected by a given fault; the degrees to which users will be affected, and methods that can be used to improve these assessments, are discussed later.

4.4 Analysis Phase

The Analysis Phase employs various techniques to optimize the dependency model search, and more quickly identify and focus on those system components that affect one or more users. The dependency model generated from the Discovery phase can be very large, and can include hundreds of resources and thousands of dependencies; furthermore, many of the resources and dependency relationships included might not have any impact on user access to programs, files and remote sites.

4.4.1 Determine Relationships with an Effect on Real Users

As an example, consider that the user names in the ps-data and lsof-file/network-data files normally include system accounts that do not represent real users. Similarly, the dependencies generated based on these system accounts might represent the dependencies for standard system services, which might not be accessed by any of the real system users.

Consequently, to optimize the dependency model search when assessing impact, we will extract only those dependencies that affect one or more real users. To determine real users, we use the contents of the w-data files. Because each w-data file only represents the users logged in at one moment in time, we use the accumulation of all w-data files we have gathered to generate a list of all real users. Then, we execute top-down search in the current dependency model for each real user to determine all of the dependency relationships (and corresponding resources) that would affect that user, and extract those relationships into an impact dependency model. This assures us that each dependency in the minimized impact model affects at least one real user.

4.4.2 Determine Relationships Most Likely to Yield Mining Results

The Analysis Phase also employs techniques to determine the dependency relationships most likely to yield significant results for data mining. The Mining phase uses fairly common algorithms to detect frequent patterns and associations in the dependency model. Generally, detecting patterns with a strong degree of confidence for a requires some minimal level of data support – if there are too few examples of a specific resource being used, then it will be difficult to detect any significant usage patterns for that resource. Though some mining algorithms provide configuration parameters (i.e. minimum support threshold) to address these issues, we take steps in the Analysis phase to remove data that unlikely to yield significant usage patterns.

More specifically, the Analysis phase first measures whether each dependency is active during each snapshot. For efficiency, it tests only those dependencies that impact one or more real system users. These results are captured in an activity matrix, which produces a summary of which dependencies are active at any moment in time. Then, for

any specific dependency, we can calculate an average amount of usage as the arithmetic mean of the number of active snapshots divided by the total number of snapshots. If this value is too low, then it is unlikely that there will be enough activity to detect any usage pattern with a significantly strong degree of confidence. Likewise, if the resource is continually being used, then it is also unlikely that significant "non-usage" patterns will be detected. Correspondingly, we establish a low-threshold and high-threshold to filter out these kinds of resources. Resources with an average usage level below the low-threshold are referred to as "sparse" resources, while resources with an average usage level above the high-threshold are referred to as "continual" resources. The remaining resources are the "frequent" resources, and their average usage levels are more likely to generate significant usage patterns.

4.4.3 Identify User-Level Programs and Resources

The Analysis phase identifies "top-level" programs, in order to focus on those programs that will be more relevant for the user. Programs invoking subprograms is captured during the Discovery phase. When programs invoke other subprograms, the user-name for the subprogram is that of the program owner. Consequently, when reviewing the output of the ps() command, the user name will be associated with the top-level program and all of the subprogram invoked to support its execution. Normally, however, the user is only aware of, and concerned with, the top-level programs. Understanding how the subprograms impact the top-level programs is important for accurately assessing the overall impact; however, the subprograms should be prioritized at a low-level when reporting impacts to the end users. Also, we do remove the subprograms during certain portions of the Mining phase, since the high-level of correlation between a top-level

program and its' subprograms can also cause difficulties for some pattern detection algorithms.

4.4.4 Identify Common Resources

The last section of the Analysis phase involves identifying "common resources." Common resources are those resources that are shared by a significant percentage of the real users, and/or by one or more programs. As an example, certain programs are used by every real user who logs on to a system; therefore, the impact of this program is independent of any specific user, and its failure will (in principle) affect any user logged on at the time of the fault. Similarly, a certain library file might be used every time a certain program is executed; therefore, this file is also independent of the user executing the program. This "user-independence" value is calculated as percentage of the occurrences that a real user executes a certain program, or a program being executed accesses a certain file or remote site. These resources are common to all users, or to all users executing a specific program. Many of these resources provide support for other directly user-accessible, top-level resources. Similarly to the subprograms mentioned above, these programs are important, but should be prioritized at a low-level when reporting impacts to the end users.

4.5 Mining Phase

The Mining Phase uses the results of the Analysis phase to detect significant patterns in the resource usage data. Specifically, we assemble and preprocess the mining data for two general scenarios: scheduled patterns, where a resource is used at certain specific times; and demand patterns, where a resource is used within a certain time frame based on the current usage states of other resources. The usage for each resource has been

captured in the Analysis phase, and the timestamps $\boldsymbol{t_i}$ have been used to capture the specific date and time information for each snapshot.

For each distinct resource in the selected set of dependencies, we calculate the tuples for scheduled and demand pattern detection. The scheduled pattern detection implemented below corresponds to search for partial periodic patterns in time-series data. The demand pattern detection that is implemented below similarly corresponds to detecting cyclic or periodic association rules, and can also be seen as an extension of autocorrelation analysis. Autocorrelation analysis is normally used in trend analysis to detect seasonal patterns by looking for correlations between each pair of i^{th} and $(i + k)^{th}$ elements in the series [39]. The mining dataset formats are given here:

$$Mining\ datasets\ for\ dependency\ \boldsymbol{d_j}\ over\ all\ collection\ periods\ \boldsymbol{t_i}:$$

$$\boldsymbol{schedule-based}: \langle \{day(\boldsymbol{t_i}), month(\boldsymbol{t_i}), date(\boldsymbol{t_i}), hour(\boldsymbol{t_i})\}, active(\boldsymbol{d_j}, \boldsymbol{t_i})\rangle$$

$$\boldsymbol{demand-based}: \langle \{active(\boldsymbol{d_k}, \boldsymbol{t_i}) | \boldsymbol{d_k} \in Dependencies\ and\ k \neq j\}, \bigvee_{\Delta \boldsymbol{t}=0}^{duration-1} active(\boldsymbol{d_j}, \boldsymbol{t_i} + \Delta \boldsymbol{t})\rangle$$

The tuples for scheduled pattern detection take the form:

$$\langle \{day(\boldsymbol{t_i}), month(\boldsymbol{t_i}), date(\boldsymbol{t_i}), hour(\boldsymbol{t_i})\}, activeResource(\boldsymbol{i}, \boldsymbol{R_j})\rangle$$

The timestamp $\boldsymbol{t_i}$ contains the day, month, date and hour attributes for the $\boldsymbol{i^{th}}$ snapshot. The label is the $activeResource(\boldsymbol{i}, \boldsymbol{R_j})$ value. The $activeResource(\boldsymbol{i}, \boldsymbol{R_j})$ value is 1 if

resource $\boldsymbol{R_j}$ was used (i.e. exists in at least one relationship of the form $\boldsymbol{R_j \rightarrow S}$ or $\boldsymbol{Q \rightarrow R_j}$) during snapshot $\boldsymbol{t_i}$, and is 0 otherwise.

The tuples for demand pattern detection take the form:

$$\langle demandAttributes, \bigvee_{dt=0}^{t} activeResource(\boldsymbol{i} + dt, \boldsymbol{R_j}) \rangle$$

The attributes are states of the other resources (excluding $\boldsymbol{R_j}$) during the $\boldsymbol{i^{th}}$ snapshot. The label is the usage for resource $\boldsymbol{R_j}$ for the periods from the $\boldsymbol{i^{th}}$ snapshot through the $(\boldsymbol{i} + \Delta\boldsymbol{t})^{th}$ snapshot, inclusive. This can be achieved simply by taking a logical-OR of the $activeResource(\ )$ values for resource $\boldsymbol{R_j}$ for each snapshot in this time period.

Once the tuples are generated, we applied fairly common data mining algorithms to generate rules for each resource. The current implementation uses the C4.5 decision tree algorithm for both the scheduled and demand pattern detection processes. We also use an iterative rule generation technique we have loosely named "uprooting" to generate multiple rule sets. When the initial tuples are fed into the C4.5 algorithm, it generates a decision tree, which determines the value of the class label based on the attributed selected for the tree with a certain confidence level. Rules are then extracted from the tree, along with the corresponding confidence level, and recorded in the appropriate rule set. Uprooting involves removing the attribute that was used as the root node of the decision tree, and re-evaluating the tuples with the now reduced attribute set. The result is a new decision tree with a new (and normally slightly reduced) confidence level. Rules are extracted from this new tree, and this process continues until the confidence level falls below a certain threshold, or we run out of attributes. Uprooting is useful in the

event we do not have the values for a certain attribute at assessment time, and are thus unable to use that attribute for predictive purposes.

We also use the activity frequency and correlation values to reduce the number of dependencies to be considered during the Mining Phase. Dependencies with a very low activity frequency will be unlikely to cause an operational impact, and will also be likely to yield trivial patterns during the mining process. Dependencies with a very high activity frequency will, on the other hand, almost certainly cause an impact; however, they will also be likely to yield trivial patterns. Consequently, dependencies with frequencies lower or higher than our established thresholds (e.g. 10% and 90%) are removed from mining consideration. We calculate the correlation value for dependency pairs that have equivalent activity frequencies, or where the difference of their activity frequencies is smaller than an established tolerance (e.g. 2%). If a pair of dependencies is strongly correlated (e.g. > 96.9%), then we can remove one of the dependencies from mining consideration.

4.6 Assessment Phase

First, we use the ***system topology*** to calculate each path from a ***failed resource*** to a user who may be impacted by the given ***technical event***. We then analyze the dependencies along each potentially impacted path. For each dependency, we use the ***system usage patterns***, ***time of failure***, ***duration***, and ***system status*** information to determine the maximum likelihood that the dependency will be active during the outage period. For each path, we use the minimum likelihood of the dependencies on the path to determine the overall likelihood that the user will be operationally impacted by the failed resource.

We remove any paths where the likelihood is less than a certain threshold, and return the remaining paths as the ***operational impact assessment***.

CHAPTER 5

DISTRIBUTED IMPLEMENTATION TECHNIQUES

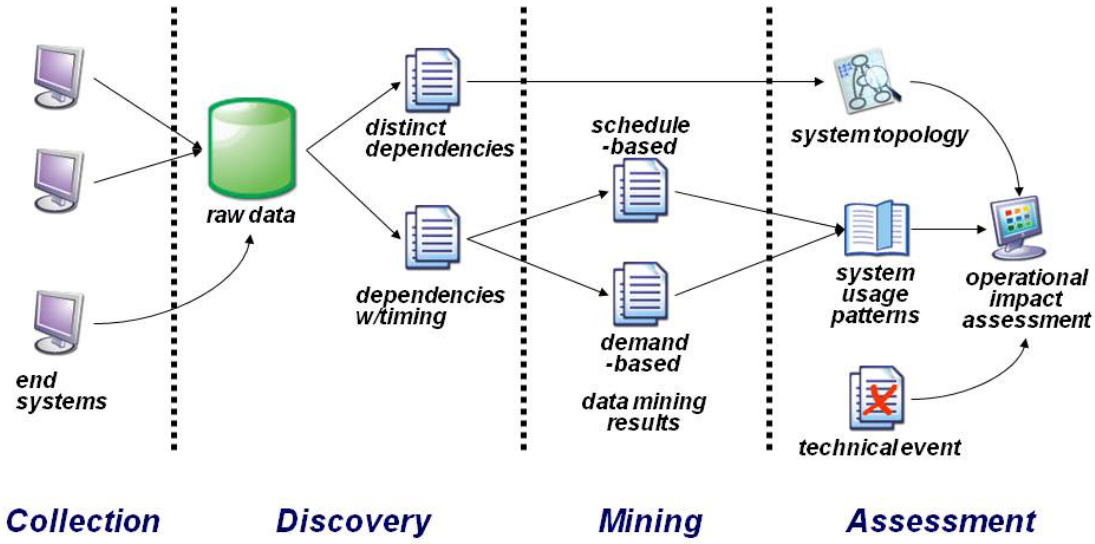

Figure 8 - Basic Dataflow Architecture

When implementing the prototype, we decided to distribute the processing associated with the Analysis Phase between the Discovery and Mining Phases. Consequently, we focused on the four phases shown in Figure 8: Collection, Discovery, Mining and Assessment. Distributing the processing on a very basic level is feasible, as demonstrated by Tang, Chang and So in the implementation of their Business-Aligned IT Service Environment (BISE) project [40]. The BISE infrastructure uses peer-to-peer (P2P) algorithms and overlay network techniques to support scalable and resilient communications.

5.1 Motivation and Overview

From a granular computing perspective, we can view the impact assessment problem in a hierarchical fashion. The enterprise system is the top level of the hierarchy, and the end systems are the lowest level granules. We can also envision intermediate levels in this hierarchy: for example, we may decide to cluster end systems that share a common local area network. This makes sense from a topological perspective, since network component faults will tend to affect the end systems in a cluster in a similar manner. We could also cluster those end systems used to support specific enterprise operations: for example, financial management, manufacturing, or inventory control. This would make sense from an operational perspective, since the end systems in these clusters will have a higher likelihood of similar usage patterns. These kinds of intermediate-level clusters are typical in large enterprises: for example, world-wide corporations often divide their resources into geographically and operationally-oriented divisions. We focus on the simpler, two-level hierarchy in my current investigations, though examining how the complexity of the hierarchy affects my results is an interesting possibility for future research.

Our experience is that the administrators have control over the end system configurations in many environments. These devices – desktops, laptops and even mobile handhelds – can normally be configured to support this kind of monitoring. Our approach assumes that each end-system uses an operating system that provides a reasonable set of diagnostic monitoring commands. We leverage the output from these commands to monitor how the end users are employing the various components in the enterprise system, and how these components interact. This is consistent with the

approach taken by Mortier, Isaacs and Barham in the Anemone project [36]. They use the end systems, as opposed to SNMP-based solutions, to collect network data. This allows them to minimize the loss of network visibility when monitoring in the presence of tunneling, encryption, dynamic port negotiation, and other modern networking techniques. We also use end systems to ensure good visibility of both local and system-wide user interactions.

Since the monitoring data is located on the end-systems, moving the impact assessment processing to the end-systems as well has the potential to minimize the amount of data transmitted across the network. This supports our design goal of supporting intermittent and low-bandwidth connectivity networks. There is a tradeoff, however: by processing the monitoring data in distributed groups, we may not detect patterns that would be discovered if we processed the data in a single, unified group. This is especially true as we mine the data to detect system usage patterns. This difference in detected patterns might affect the accuracy of the impact assessments. This tradeoff is similar to the principle of exploiting the tolerance for imprecision to achieve tractable and low cost solutions as proposed in [41]. We believe that it is possible to distribute, either partially or fully, the impact assessment processes without significantly reducing the accuracy and overall quality of the assessment results.

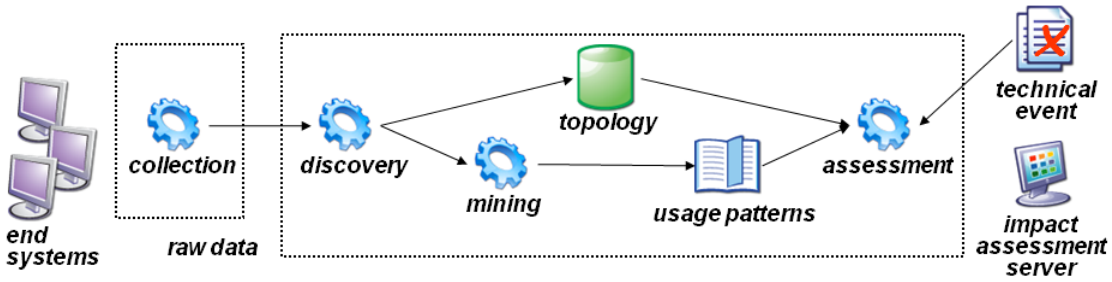

Figure 9 - Centralized Assessment Processing

5.2 Explanation of the Different Distribution Approaches

Given that the monitoring data is collected from the end systems, it is natural to consider the possibility of minimizing the transmission of the data to a centralized location. We consider the implementation of my impact assessment system using three distinct approaches: centralized, partially distributed, and fully distributed. In all approaches, the Collection phase is conducted at the end-systems. In the centralized approach shown in Figure 9, all of the collected data is sent to the impact assessment server. We then perform the Discovery, Mining and Assessment phases entirely at the server.

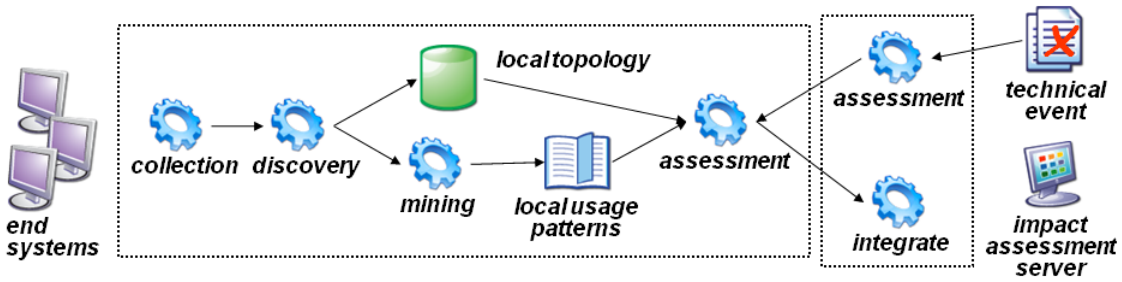

Figure 10 - Fully Distributed Assessment Processing

By contrast, in the fully distributed approach shown in Figure 10, the Discovery and Mining phases are conducted exclusively at each end system – no dependency data is

forwarded to the server. Queries issued during the Assessment phase are forwarded to, and processed by, each end-system. The results from all end-systems are returned to the server, where they are assembled to form the overall assessment result.

The partially distributed approach shown in Figure 11 is the most complicated of the three approaches, and attempts to leverage the strengths of centrally and fully distributed processing. In the partially distributed approach, the Discovery phase is conducted at each end system. The dependencies are divided into two groups, based on the resource zones: *local* and *global*. Local dependency information is maintained at each end-system, while all global dependency information is sent to the impact assessment server. The Mining phase occurs on the end-systems and the server.

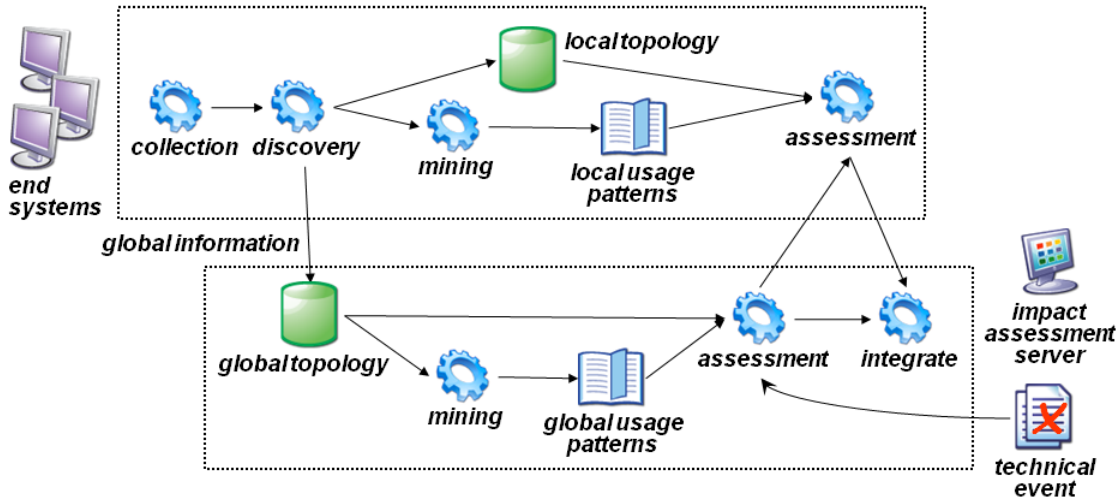

Figure 11 - Partially Distributed Assessment Processing

Finally, queries during the Assessment phase are started at the server. If local components are encountered during the assessment process, then that component information is sent to the corresponding end-systems, and the assessment process is

executed on those end-systems as well. The results from each end-system are returned to the server, and combined to form the overall assessment result.

CHAPTER 6

SYSTEM ARCHITECTURE & IMPLEMENTATION

6.1 Architecture and Technical Overview

The main effort of our most recent research has been to develop a more complete and comprehensive prototype of our system. In our initial research, we automated the processes within each of the four phases – Collection, Discover, Mining and Assessment – but data transfer between the different phases was conducted mainly by manual means. In developing and implementing the Impact Assessment System Architecture as shown in Figure 12, we first established a more common set of tools and languages for our system. The architecture shown is not complete – for clarity, it does not show all of the programs being used. The system uses a total 41 Perl programs of varying size, and the architecture displays the main programs used by the administrators to collect data, and to execute the assessment processing. Similarly, the database actually uses over 30 different tables to store data and support temporary processing. We also use SQL scripts to allow the Perl programs to interact with the database. There are 32 persistent scripts defined; and, three of the Perl programs also generate SQL scripts dynamically to be used for that specific invocation of the program.

Since we were working actively with the Georgia Tech Research Network Operations Center (GT-RNOC), we decided to use the Perl as a common language [42]. They were already using Perl for a number of projects, and the language also provided a number of features – for example, regular expression processing, and the straightforward

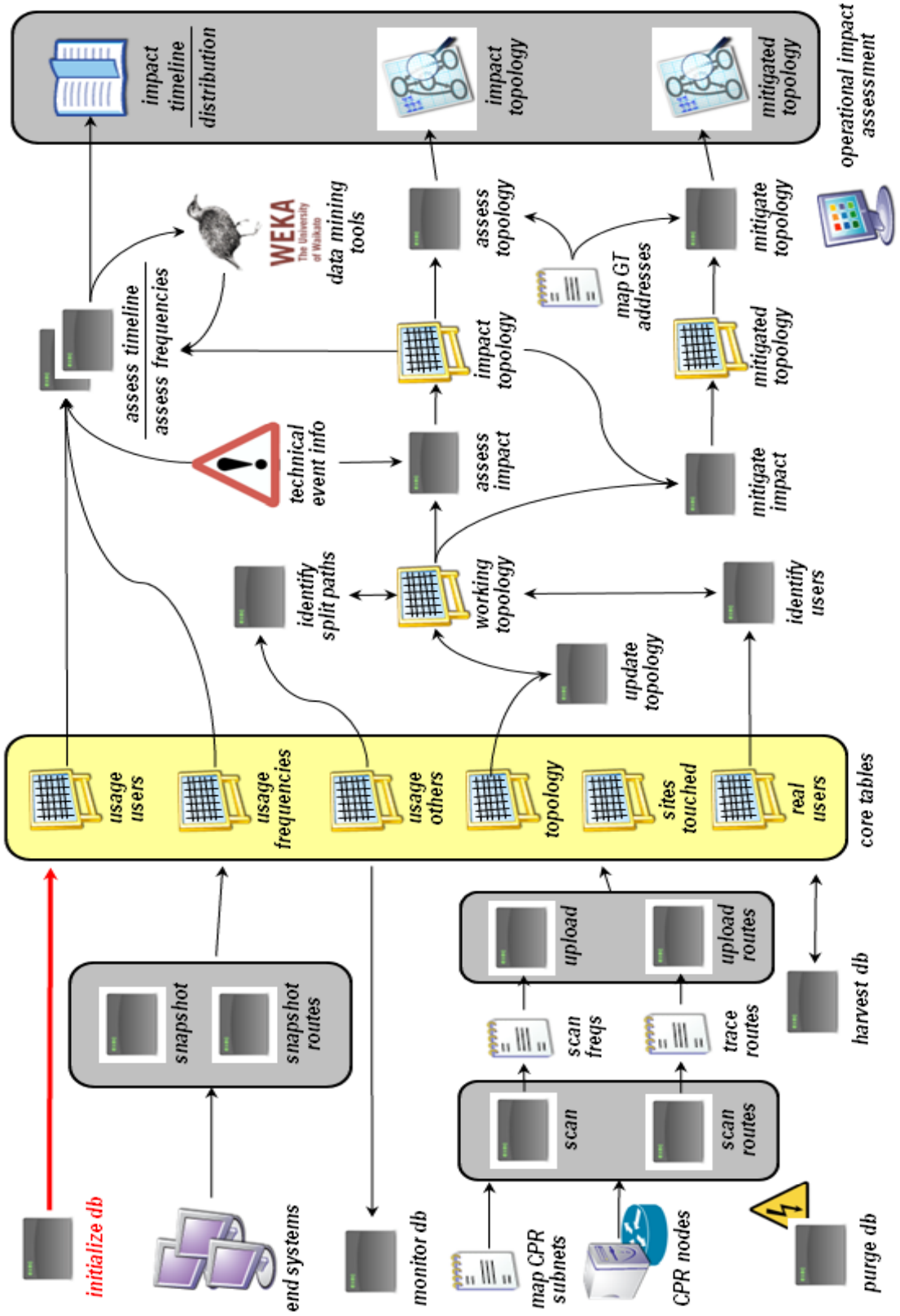

Figure 12 – Operational Impact System Technical Architecture

ability to invoke operating system commands – that were ideal for the types of data processing that we were performing.

As part of our efforts to make the system portable, and to conserve system resources as much as possible, we decided to use the Apache Derby database [43]. The Derby database is based on the IBM Cloudscape database, and offers a basic level of SQL compliance. More importantly, it has been designed as a small-footprint database written entirely in Java, and can run on the Java Virtual Machine, which increases the number of platforms on which it can be executed without extensive administration and pre-installation overhead. This supports our efforts to implement and test our system in a distributed mode. We also leverage the open-source WEKA toolkit to support our data mining requirements, such as generating schedule- and demand-based decision trees for assessing impact, and finding clusters for dependencies with similar activity characteristics [44]. Finally, we use the Graphviz application to render the impact and mitigated topology diagrams [38].

When we are ready to assess the operational impact for a specific event, we collect the ***technical event information***: the failed resources, the duration of the failure, and the time range over which we wish to assess the impact. The basic process is first to assess the topology, in order to determine which user-based dependencies might be affected by the failed resources. Then we use the timing information for those dependencies, along with the failure duration information, to generate a model to predict the likelihood of activity for each dependency at any given time. Finally, we evaluate the each of the potentially impacted dependencies over the designated range of time to generate a representation of the impact likelihood timeline. In the next few chapters, we

will take a look at some of the key processes that we use in the latest version of our architecture.

6.1.1 Continuous Data Collection

The system is designed to process the data in a pipeline-like format. Data is collected from end systems, CPR nodes, and other sources on a regular basis. Traceroute data is collected using the ***snapshot_routes( )***, ***upload_routes( )*** and ***scan_routes( )*** procedures; other raw operating system data is collected using the ***snapshot( )***, ***upload( )*** and ***scan( )*** procedures. The separate procedures are designed to collect traceroute data at a different rate, since invoking the traceroute command has an impact on the network as well as the local machine. The snapshot_routes() procedure tracks how recently the traceroute was executed on that machine to each specific site, and then stores that information in the ***sites_touched table***. The snapshot_routes() procedure then uses a round robin technique to rotate through the identified sites in order of access frequency, and to ensure that there is as much coverage of the entire network as possible. The basic snapshot algorithm is given here:

Algorithm: Capture Local Operating System Data

timeStamp := current date and time based on the system clock;
execute "who" command ("w –h") & **store** results in the *whoDump* file;
execute "process" command ("ps -eo uid,user,pid,ppid,comm") & **store** results in the *processDump* file;
execute "disk free" command ("df -ah") & **store** results in the *deviceDump* file;
execute "lsof" command ("lsof") & **store** results in the *lsofDump* file;
machineName := (local) identifier for the system on which process is being executed;
for each *line* in the *whoDump* file **do**
    parse *line* into components: ⟨*userName*, *programName*⟩;
    generate output record & **store** in the *dependencyTopology* file:
        ⟨*timeStamp*, global | user | *userName*, *machineName* | program | *programName*⟩;
    generate output record & **store** in the *dependencyTopology* file:
        ⟨*timeStamp*, global | user | *userName*, global | origin | *machineName*⟩;

    store ⟨*timeStamp*, *userName*⟩ in the *realUser* file;
end for
initialize/empty the *processName*[] and *processParent*[] arrays;
for each *line* in the *processDump* file **do**
    parse *line* into components:
        ⟨*userName*, *processID*, *parentProcessID*, *programName*⟩;
    generate output record & **store** in the *dependencyTopology* file:
        ⟨*timeStamp*, global | user | *userName*, *machineName* | program | *programName*⟩;
    processParent[*processID*] := *parentProcessID*;
    processName[*processID*] := *programName*;
end for
for each *processParent*[*processID*] that is defined **do**
    if (*processName*[*processParent*[*processID*]] is defined) **then**
        parent := *processName*[*processParent*[*processID*]];
        child := *processName*[*processID*];
        generate output record & **store** in the *dependencyTopology* file:
            ⟨*timeStamp*, *machineName* | program | *parent*, *machineName* | program | *child*⟩;
    end if
end do
initialize/empty the *deviceZone*[] array and *devicePriority*() list;
for each *line* in the *deviceDump* file **do**
    parse *line* into components: ⟨*deviceLocation*, *mountPoint*⟩;
    if (*deviceLocation* represents an IP(v4) address) **then**
        remoteAddress := **extract** IP address from *deviceLocation*;
        generate output record & **store** in the *dependencyTopology* file:
            ⟨*timeStamp*, global | device | *deviceLocation*, global | site | *remoteAddress*⟩;
        deviceZone[*mountPoint*] := “global”;
    else
        deviceZone[*mountPoint*] := *machineName*;
    end if
    append *mountPoint* to *devicePriority*() list;
    zone := *deviceZone*[*mountPoint*];
    generate output record & **store** in the *dependencyTopology* file:
        ⟨*timeStamp*, *zone* | directory | *mountPoint*, *zone* | device | *deviceLocation*⟩;
end for
sort the elements of *devicePriority*() in order of descending element length;
for each *line* in the *lsofDump* file **do**
    parse *line* into components: ⟨*programName*, *userName, descriptor, type, fileName*⟩;
    generate output record & **store** in the *dependencyTopology* file:
        ⟨*timeStamp*, global | user | *userName*, *machineName* | program | *programName*⟩;
    if (*descriptor* represents a text or character file) **then**
        scan & **locate** the first *mountPoint* (in order) that is contained within *fileName*;
        shortName := **extract** basic file name (remove path information) from *fileName*;
        zone := *deviceZone*[*mountPoint*];
        generate output record & **store** in the *dependencyTopology* file:

⟨*timeStamp*, global | user | *userName*, *zone* | file | *shortName*⟩;
generate output record & **store** in the *dependencyTopology* file:
⟨*timeStamp*, *machineName* | program | *programName*, *zone* | file | *shortName*⟩;
generate output record & **store** in the *dependencyTopology* file:
⟨*timeStamp*, *zone* | file | *shortName*, *zone* | directory | *mountPoint*⟩;
end if
if (*type* represents an IP(v4) address) **then**
parse *fileName* into components: ⟨*localPort*, *remoteSite, remotePort*⟩;
generate output record & **store** in the *dependencyTopology* file:
⟨*timeStamp*, global | user | *userName*, global | site | *remoteSite*⟩;
generate output record & **store** in the *dependencyTopology* file:
⟨*timeStamp*, *machineName* | program | *programName*, global | site | *remoteSite*⟩;
generate output record & **store** in the *dependencyTopology* file:
⟨*timeStamp*, global | site | *remoteSite*, *machineName* | port | *remotePort*⟩;
generate output record & **store** in the *dependencyTopology* file:
⟨*timeStamp*, *machineName* | program | *programName*, *machineName* | port | *localPort*⟩;
end if
end for

The raw data is collected and processed in the format:

$\langle date\_time\_stamp, resource_1, resource_2 \rangle$

where $resource_1$ is dependent on $resource_2$, and each resource is fully qualified by the triple $\langle zone, type, identifier \rangle$. The zone attribute is mainly significant when discussing distributed assessment techniques; consequently, we will occasionally represent a resource using the "shorthand" 2-tuple $\langle type, identifier \rangle$ when the zone attribute value is not significant. The data is stored in the database, such that user-based dependencies of the form (*user* | $U \rightarrow$ *resource* | *R*) are stored in the ***usage_users*** table; and all other dependencies are stored in the ***usage_others*** table. The timing information is used when assessing timelines, and identifying redundant (transitive) dependencies; it is not required when assessing topological impacts. Consequently, we extract the dependencies from the

resource and timing information in the usage_users and usage_others tables, and store the combined information in the ***topology*** table.

6.1.2 Collecting and Representing Traceroute Data

The traceroute data is taken for those web and remotely accessed sites for which there is some measurable user demand. For snapshot_routes(), the system tracks those sites that have been requested by one or more users, and then uses some metric to collect traceroute data for some subset of the sites. The metric currently used is to rank the sites in terms of the number of times that they have been accessed over the most recent period, and then to collect data for the most frequently accessed sites. We collect data for a relatively small subset of the sites (as opposed to the entire population) to minimize the impact on the network. In contrast, the upload_routes() procedure has to determine which sites should selected for tracerouting, and receives bulk data from Netflow logs, which record information on every site that has been accessed over a certain time period. In this case, we employ filtering methods like lossy counting to identify the most frequently sites, and then transmit that information to the upload_routes() processes that have been instantiated on various CPR nodes distributed across the Georgia Tech network.

The snapshot_routes() and upload_routes() procedures translate traceroute data into resources represented in the dependency topology. The main difference between the two procedures is their data sources: the snapshot_routes() procedure executes the traceroute command on the local end system, while the upload_routes() procedure receives data from router logs, CPR nodes, and other resources distributed across the system we are monitoring. Otherwise, the two procedures perform the same fundamental process, as shown here:

Algorithm: Capture Traceroute Data

timeStamp := current date and time based on the system clock;
execute "traceroute" command ("traceroute -n -q 1 -m 30 ⟨*remoteSite*⟩")
    & **store** results in the *tracerouteDump* file;
machineName := (local) identifier for the system on which process is being executed;
previousType := "origin";
previousHop := *machineName*;
traceState := "regular";
for each *line* in the *tracerouteDump* file **do**
    if (*line* represents an invalid IP address/∗ and *traceState* = "regular") **then**
        traceState := "cloud";
    else if (*line* represents a valid IP address)
        nextHop := **extract** IP address from *line*;
        if (*traceState* = "cloud") **then**
            cloudHop := **concatenate** "∗" and *nextHop*;
            generate output record & **store** in the *dependencyTopology* file:
                ⟨*timeStamp*, global | router | *cloudHop*, global | *previousType* | *previousHop*⟩;
            previousType := "router";
            previousHop := *cloudHop*;
            traceState := "regular";
        end if
        generate output record & **store** in the *dependencyTopology* file:
            ⟨*timeStamp*, global | router | *nextHop*, global | *previousType* | *previousHop*⟩;
        previousType := "router";
        previousHop := *nextHop*;
    end if
end for
if (*traceState* = "cloud") **then**
    cloudHop := **concatenate** "∗" and *remoteSite*;
    generate output record & **store** in the *dependencyTopology* file:
        ⟨*timeStamp*, global | router | *cloudHop*, global | *previousType* | *previousHop*⟩;
    previousType := "router";
    previousHop := *cloudHop*;
end if
generate output record & **store** in the *dependencyTopology* file:
    ⟨*timeStamp*, global | router | *remoteSite*, global | *previousType* | *previousHop*⟩;

As an example, consider Figure 13. The user U has accessed two different sites:

www.biz.demo, and www.fun.demo. The user accesses the sites using the computer

system M. The traceroute data is displayed sequentially by hop number, beginning with

the first hop away from system M, including all of the intermediate hops from M to the destination site, and ending when the site is reached.

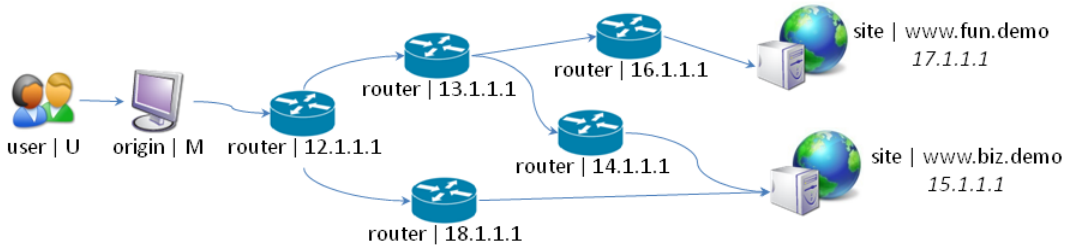

Figure 13 - Sample Traceroute Paths

The data is normally presented as IP addresses representing the devices at each hop, and we generally avoid using the name resolution features to minimize the use of computing resources. We have generated and stored the DNS name mappings for certain IP addresses, however, to make the dependency topologies more clear and readable. Also, the IP address for the devices at one or more hops might not be returned; for example, by a device configured not to respond to ICMP traffic for security reasons. In these cases, a star ('*') is normally returned in lieu of an IP address.

The snapshot_routes() and upload_routes() procedures basically reverse the route from M to the destination sites for clarity: our main focus in assessing operational impact is understanding which resources are needed by the user, and how those user-required resources are affected by the other system resources. The resources that are "directly required" by the user are the data and services provided by the remote sites. The other resources, like the computing system and browser used to access the sites, and the intermediate routing devices, are only there to provide the user a means to access the remote sites. The "reversed traceroute" representation allows us to better emphasize and

display these relationships. Also, although not shown in these diagrams, hops that have unrevealed IP addresses are treated as "clouds" between the closest recognizable IP addresses at the points of entry and exit. This keeps the resulting topology as compact and readable as possible, while maintaining accurate dependency information for impact assessment. Our representation of the network paths shown in Figure 13 is given in the following Figure 14.

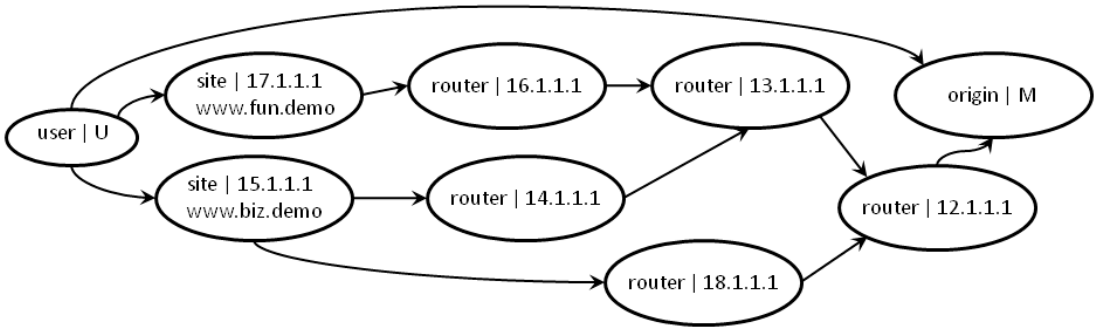

Figure 14 - Impact Assessment Representation of Traceroute Paths

As an example, suppose that the router 13.1.1.1 has failed. The potential impact is assessed by the ***assess_impact( )*** procedure. The potential operational impact is that the *user U* will be unable to access the sites www.biz.demo and www.fun.demo, which is determined by calculating the transitive closure of those sites that are dependent on the failed router.

Similarly, our recent focus on the networking subset of the dependency topology model led us to develop a similar procedure called ***mitigate_impact()***. This procedure leverages the results of the assess_impact() procedure to determine if there are alternate paths to any of the potentially impacted sites. It determines this by first extracting the set of users from the potentially impacted (*user* | *U* → *site* | *S*) dependencies. For each of the

users, the procedure then identifies all of the (*user* | $U \rightarrow$ *origin* | *M*) dependencies, to extract the set of alternate starting points for the next calculation. Intuitively, we need to determine the different access points that the impacted users could use to access their remote sites. The final step determines the sites that are accessible via an alternate path by calculating the transitive closure of those sites that are dependent on the one of the alternate access points/origins M. During the transitive closure calculations, the process eliminates any paths that attempt to traverse the failed router (or any other failed resources).

6.1.3 Filtering & Assessing the Topology

To assess the topology, we begin by generating the most current topology information with ***update_topology( )*** procedure. This loads the comprehensive and unique topology information into the ***working_topology*** table. The ***identify_users( )*** procedure leverages the information in the ***real_users*** table to identify the subset of the topology that supports real users, as opposed to "system-based" users like background processes. The ***identify_split_paths( )*** procedure uses the timing information in the usage_others table to determine and remove redundant dependencies for resources that are accessed concurrently by users and programs. Next, we execute the ***assess_impact( )*** procedure, which uses the contents of the working_topology table to identify the subset of dependencies that are affected by the failed resources, and stores the results in the ***impact_topology*** table. Finally, we execute the ***assess_topology( )*** procedure to generate a DOT-formatted (Graphviz-viewable) image of the ***impact_topology***. The assess_topology() procedure also leverages ***DNS_map*** information to add domain names for some IP addresses, to make the resulting image more understandable.

The ***identify_users( )***and ***identify_split_paths( )*** procedures are used to reduce the size of the working topology, to improve readability and reduce the computing resources needed for assessing impact. The identify_users() procedure leverages the data collected by the w() operating system command, and this data is used to distinguished between those computer accounts that represent actual human users, versus those accounts used by operating system services, daemons, background processes, etc. Once the real user accounts are identified, we calculate the transitive closure of the resources on which the real user accounts depend. Intuitively, this subset of the entire dependency topology includes only those resources that could generate an operational impact on at least one real user. Any dependencies that do not belong to this subset are removed from the current instantiation of the working topology, but not from the complete set of dependencies within the database. This gives us the flexibility to assess impact on different level: for example we could assess the overall impact on the system, including OS services, background processes, etc. Then, we could run the identify_users() process, and re-execute the assessment to focus our analysis only on real users.

The identify_split_paths() procedure is used to distinguish "user-level" resources – for example, files, sites and programs – that are used only to support the execution of applications and services. The intent is to associate these user-level resources directly with the users, and to associate the other resources directly with the application they are used to support. The difficulty occurs when extracting information from the lsof() operating system data – all open files are associated with both the owning user's ID and the process ID. If this data is entered directly into the discovery topology, it generates potentially redundant and unnecessary dependencies. This redundancy takes the form of

(*user* | $U \rightarrow$ *program* | $P \rightarrow$ *resource R*, and *user*| $U \rightarrow$ *resource* | *R*), where resource R is directly associated with user U, and also indirectly associated with user U via program P. The identify_split_paths() algorithm is given here:

Algorithm: Identify Split Paths

for each combination of (*resourceU*, *resourceP*, *resourceS*) **do**
    times[$U \rightarrow S$] := the set of times where *resourceU* $\rightarrow$ *resourceS* is active;
    times[$U \rightarrow P$] := the set of times where *resourceU* $\rightarrow$ *resourceP* is active;
    times[$P \rightarrow S$] := the set of times where *resourceP* $\rightarrow$ *resourceS* is active;
    if (*times*[$U \rightarrow S$] = 0 or *times*[$U \rightarrow P$] = 0 or *times*[$P \rightarrow S$] = 0) **then**
        skip to the next combination of resources;
    end if
    times[*indirect*] := *times*[$U \rightarrow P$] $\cap$ *times*[$P \rightarrow S$];
    times[*concurrent*] := *times*[$U \rightarrow S$] $\cap$ *times*[*indirect*];
    concurrentRatio := | *times*[*concurrent*] | / | *times*[$U \rightarrow S$] |;
    if (*concurrentRatio* > *ratioThreshold*) **then** {
        delete/remove the *resourceU* $\rightarrow$ *resourceS* dependency;
    else
        delete/remove the *resourceP* $\rightarrow$ *resourceS* dependency;
    end if
end for

As a practical example, consider the case of a user working with a popular word processing program. The user is likely to use the word processor to edit a certain report file; our intent is to associate this report file directly with the user only, and remove the dependency from program on this report file. On the other hand, there may also be a template file that is opened by the program to support normal functionality; our intent is to associate this template file directly with the program only, and remove the direct dependency from the user on this template file.

We determine whether to associate the resource directly with the user or program by analyzing the activity frequencies between the three elements. If there is a reasonably strong positive correlation between the activity levels for the resource and the program (i.e. correlation > 90%), then we associate the resource directly with the program;

otherwise, we associate the resource with the user. Executing the assess_impact() procedure generates the list of (*user* | $U \rightarrow$ *resource* | *R*) dependencies that could potentially be impacted by the designated technical event. To better assess the potential operational impact, we execute the ***assess_timeline( )*** procedure to determine the likelihood that each of the dependencies that have been identified would be active during the timeframe of the technical event.

6.1.4 Assessing the Timeline

Once the impacted dependencies have been determined, we execute the assess_timeline() procedure to determine the likelihood that the user-based dependencies would actually be active during the resource failure period. For each user-based dependency in the impact_topology table, we extract the timing information from the usage_users table. The assess_timeline() procedure processes and formats the data, and then calls ***data mining routines in the WEKA toolkit*** to generate one decision trees that will predict the impact likelihood for each the of the dependencies. The WEKA routine used actually generates an equivalent ruleset for each decision tree, which allows the assess_timeline() procedure to evaluate each tree over each minute of the designated time range. The information represents the ***impact_timeline***, and is then displayed in a text-based format. We have also generated some scatter graph based views of this information to better help administrators see the activity trends for a given technical event. The translation of text data to the more visual scatter graph format is currently manual, but could reasonably be automated in future versions of the system.

6.1.5 Assessing Mitigated Impact with the Network Topology

While working with the GT-RNOC, we focused more on the network subset of our dependency topology model. We extended the functionality of the overall system to provide more options for the users. One of the extensions was the capability to determine if the impact on (*user* | $U \rightarrow$ *site* | *S*) dependencies would be mitigated by having alternate paths to the destination site. The ***mitigate_impact( )*** procedure uses information from the working_topology and impact_topology tables to determine if there are any relevant alternate paths, and stores those results in the ***mitigated_topology*** table. The ***mitigate_topology( )*** procedure can then be used to generate a DOT-formatted representation of the dependency topology with both failed and alternate dependency links. Similarly, much of the Netflow data that we collected was processed using lossy-counting techniques, which yielded approximate occurrence frequencies for each of the impacted dependencies. We store this information in the ***usage_frequencies*** table, and the ***assess_frequencies( )*** procedure uses this information to produce an ***impact_distribution***. The assess_frequencies() procedure operates similarly to the assess_timeline() procedure; it differs by using the average frequencies to determine impact likelihood, whereas assess_timeline() uses simpler discrete activity measurements (i.e. either the dependency is active, or it's inactive) to determine the impact likelihood.

6.1.6 Support Operations

Some procedures are used to support the assessment operations. The ***initialize_db( )*** procedure is used to create the Derby database structures, such as the core table, views and indexes needed to store data. The ***monitor_db( )*** procedure will display some common statistics about the current state of the assessment database, such as the number

of rows, and distribution of values for many of the tables. As data is collected, it may become necessary to delete some of the older data to ensure quick and consistent response times. The ***purge_db( )*** procedure will remove all data from the core tables, and is especially useful for experimenting with different datasets. The ***harvest_db( )*** procedure, in contrast, archives older data into an external file, and ensures that the number of rows in each of the core tables is lower than a preset limit for that table. The harvest_db() procedure is intended for use in production environments, where the most recent data is maintained in the current system, and older data can be reloaded as required, or off-loaded into a separate system for more extensive analysis.

6.2 Key Algorithms

6.2.1 Lossy-Counting Based Log Scanning

In the Collection Phase, we are required extract key activity data from operating system and networking log files. Our goal of providing near real-time assessments means that we need to be able scan these very large files very quickly. There are a number of algorithms that have been developed to scan large files in this manner, such as the lossy counting algorithm for identifying frequent items within a data stream [39]. We began by applying a basic lossy-counting algorithm to our log files, which were ordered chronologically as a data stream. We encountered problems when the log files contained duplicate records. Having two or more records with the exact same dependency and time values does not add any provide any additional information during the Discovery or Mining Phases. In fact, the presence of duplicate records for a specific element artificially inflated the frequency count for that element, and similarly reduced the approximate frequency count for other non-repeating elements. This frequency distortion

adversely affected our ability to identify the elements (e.g. users and resources) that would most likely be impacted. To overcome these challenges, we employed a log-scanning algorithm that is based on the principles used in the basic lossy-counting algorithm, but modified to compensate for duplicate records and bursty traffic patterns.

More specifically, we consider each log record to be an element in the data stream, where the records are generally in the format *active*(*dependency d*, *time t*). In the basic lossy-counting algorithm without duplicates, each occurrence of a specific element in the stream would be included in the frequency count for that element. In our case, however, we are interested in assessing the impact on each dependency at different times. Consequently, our goal is to approximate the frequency count for *dependency d*, as opposed to the pair (*dependency d*, *time t*). Then, we will extract the specific timing (i.e. the unique values for *time t*) or frequency information for *d* if, and only if, the approximate frequency for *d* meets a certain established threshold. Our modified lossy-counting algorithm is shown here:

Algorithm: Lossy-Counting Based Log Scanning

[for error bound (ε), minimum support (σ)]

---

bucketWidth := $\left\lceil \frac{1}{errorBound(\varepsilon)} \right\rceil + 1$;
itemCount, *bucketCount* and *timeBoundary* := 0;
bucketNumber := 1;
initialize/empty the *frequency*[] (f), *deltaError*[] $(\Delta\varepsilon)$, and *timeCheck*[] arrays;
for each *line* in the *bulkData* file **do**
    parse *line* into components: (*timeStamp*, *sourceAddress*, *destinationAddress*);
    if (*sourceAddress* is within a GT IP subnet) **then**
        localAddress := *sourceAddress*;
        remoteAddress := *destinationAddress*;
    else
        localAddress := *destinationAddress*;
        remoteAddress := *sourceAddress*;
    end if
    if (*localAddress* is a known CPR node or site) **then**

```
        netConnection := ⟨nameCPR(localAddress), remoteAddress⟩;
    else
        netConnection := ⟨nameCPR("unknown"), remoteAddress⟩;
    end if
    if (timeCheck[netConnection] = timeStamp) then
        skip to the next line in the bulkData file;
    end if
    if (frequency[netConnection] is undefined or frequency[netConnection] ≤ 0) then
        frequency[netConnection] = 1;
    else
        frequency[netConnection] = frequency[netConnection] + 1;
    end if
    deltaError[netConnection] := bucketNumber – 1;
    timeCheck[netConnection] := timeStamp;
    if (timeStamp ≠ timeBoundary) then
        timeBoundary := timeStamp;
        itemCount := itemCount + 1;
        bucketCount := bucketCount + 1;
    end if
    if (bucketCount ≥ bucketWidth) then
        for each frequency[value] that is defined do
            if (frequency[value] + deltaError[value] ≤ bucketNumber) then
                remove/undefine value;
            end if
        end for
        bucketNumber := bucketNumber + 1;
        bucketCount := 0;
    end if
end for
threshold := itemCount (minSupport(σ) – errorBound(ε));
for each frequency[value] that is defined do
    if (frequency[value] ≥ threshold) then
        store value for future reference
    else
        remove/undefine value;
    end if
end for
```

In the basic lossy-counting algorithm, we use the attribute variables *frequency*(*d*) and *delta_error*(*d*) to calculate the approximate frequency for dependency *d*. To avoid counting duplicates, we also employ the additional attribute variable *time_check*(*d*) to record the most recent time value for which *d* was active. We process the log file records

in increasing chronological order. Occasionally, we have noticed the anomaly that the time values within a single log file might appear out of order for a very small (normally less that 1%) number of records compared to the total size of the file, but this disordering has not caused any significantly adverse effect on our results.

When processing a new record *active*(*dependency d*, *time t*), we compare the value *t* with *time_check*(*d*) to ensure that the new record is not a duplicate. If *t* > *time_check*(*d*), then we update *frequency*(*d*), *delta_error*(*d*) and *time_check*(*d*) in accordance with the lossy-counting algorithm. Otherwise, the record is a duplicate: consequently, we discard that record, and continue by scanning the next record in the log file. Also, since we are discarding records, we must also reconsider how we determine the bucket boundaries. In the basic lossy-counting algorithm, the bucket width, *w*, is determined by the desired error bound, ε. Since each element in the data stream is included in the frequency count, then a bucket boundary is reached every $w = \left\lceil \frac{1}{\varepsilon} \right\rceil$ elements.

In one sense, this situation corresponds to the arrival of one element per time period. In the case of bursty traffic, however, multiple dependency values *d* can occur during the same time *t*. Even if an estimate average frequency of dependencies per time period (*s*) is determined, the number of dependency values per time period could still vary widely (i.e. the variance of *s* could still be very large). We believe that a significantly large variance for *s* can lead us to underestimate the approximate frequency for some dependencies if we simply use the basic lossy-counting algorithm. To compensate for this effect, we process the set of dependencies that occur during a specific time period as if they were as single element; consequently, we count distinct time

periods as elements as instead of individual records, and we reach a bucket boundary every *w* time periods as in the normal algorithm. Unlike the normal algorithm, however, we maintain frequency information for each distinct dependency, as opposed to maintaining frequency information for the set as a whole. The frequency adjustments at each bucket boundary are then performed as in the basic lossy-counting algorithm: each *dependency d* is evaluated, and removed from the list if:

$$frequency(d) + error\_bound(d) < b$$

where *b* is the current bucket number. From the perspective of each individual dependency, the approximate frequency for that dependency more accurately represents the frequency that the dependency occurs over time, less impacted by duplicate records and bursty traffic.

6.2.2 Producer-Consumer Approach for Impact Windows

In the Mining Phase, we are required to calculate the likelihood of a dependency being impacted at a given time. Our general approach is to use the collected usage data to construct a decision tree, where the tree uses splitting nodes based on the time components (e.g. date, hour, minutes, day of the week) and related dependency attributes at the time of the technical impact, and the leaf nodes designate the likelihood of an operational impact at that time. Our original model was designed to collect data at the end systems (e.g. workstations, laptops), where the snapshot() program would be configured to capture OS data at specific intervals. When computing the usage data needed to generate the decision tree, the estimated impact for a specific time t_i is calculated as some activity function of the activity values for *dependency d* between

times t_i and $(t_i + \Delta t)$, where Δt represents the expected duration of the technical event (e.g. resource failure or planned maintenance outage).

Our later implementations also leveraged log file data from other sources, such as network routers. We realized that the log file data collected from Netflow router logs, for example, was not necessarily collected over a uniform timeline. The non-uniform timeline requires us to modify our activity function. With a uniform timeline, we can use a fixed-size sliding window of activity values to perform the activity function calculations. With a non-uniform timeline, we use a variable-size sliding window of activity values, and we compute the difference between adjacent times to ensure that we have a window that is at least as wide as the expected failure duration. The algorithm we use is shown here:

Algorithm: Producer-Consumer Activity

timeIndex := 0;
currentTimeStamp := 0;
initialize/empty the *intervalTable*[] and *activityGrid*[][] arrays;
intervalTable[0] := 0;
for each *line* in the *tracerouteDump* file **do**
    parse *line* into components: ⟨*nextTimeStamp*, *resourceA*, *resourceB*⟩;
    connection := *resourceA* → *resourceB*;
    if (*nextTimeStamp* > *currentTimeStamp*) **then**
        intervalTable[*timeIndex*] := (*currentTimeStamp* – *nextTimeStamp*) in minutes
        currentTimeStamp := *nextTimeStamp*;
        timeIndex := *timeIndex* + 1;
    end if
    activityGrid[*timeIndex*][*connection*] := 1;
end for
zeroize/set to zero all empty/undefined entries in the *activityGrid*[][] array;
totalInterval := 0;
producer := 0;
consumer := 0;
initialize/empty the *windowTable*[], *durationTable*[] and *impactGrid*[][] arrays;
while (*producer* < *timeIndex*) **do**
    while (*producer* < *timeIndex* and *totalInterval* < *impactDuration*) **do**
        for each *link* in the *impactDependencies* list **do**
            windowTable[*link*] := *windowTable*[*link*] + *activityGrid*[*producer*][*link*];

```
        end for
        totalInterval := totalInterval + intervalTable[producer];
        producer := producer + 1;
    end while
    while (totalInterval ≥ impactDuration) do
        for each link in the impactDependencies list do
            impactGrid[consumer][link] := windowTable[link];
            windowTable[link] := windowTable[link] − activityGrid[consumer][link];
        end for
        durationTable[consumer] := totalInterval;
        totalInterval := totalInterval − intervalTable[consumer];
        consumer := consumer + 1;
    end while
end while
```

As an example, earlier implementations of our system include a "binary" activity function, which produces a "1" if any of the activity values is greater than 0, and "0" otherwise. Other activity functions produce the maximum and average activity values for that period. The set of records:

$$\{\langle time\ \boldsymbol{t_i}, activityFunction(\boldsymbol{S}(\boldsymbol{t_i}, \Delta\boldsymbol{t}))\rangle \mid for\ all\ collected\ records\}$$

$$where\ \boldsymbol{S}(\boldsymbol{t_i}, \Delta\boldsymbol{t}) = \{\langle \boldsymbol{t_j}, activity(\boldsymbol{t_j})\rangle \mid \text{where } \boldsymbol{t_i} \leq \boldsymbol{t_j} \leq \boldsymbol{t_i} + \Delta\boldsymbol{t}\}$$

is then used to generate the decision tree which will be used during the ensuing Assessment Phase. When collecting data from the end systems, we can control the interval at which we execute the snapshots, which allows us to easily determine the number of activity values needed for the activity function. For example, if we are using a fairly common snapshot interval of five minutes, then an expected failure duration of 1 hour would require that we examine 12 activity values for each time $\boldsymbol{t_i}$, from t_i minutes, $(t_i + 5)$ minutes, and so on through $(t_i + 55)$ minutes.

The procedure for producing records for generating the decision tree alternates between two basic cycles. As an example, suppose that we have an expected failure

duration of one hour, such that $\Delta t = 60$, since we measure failure durations in minutes. In the first cycle, given a starting time of $\boldsymbol{t_i}$, we continue to scan the successive times t_{i+1}, t_{i+2}, etc. until we find t_{i+k} such that $(t_{i+k} - t_i) \geq \Delta t$. This also ends the first cycle (for now), and begins the second cycle of the procedure. At this point, we use the activity value pairs from $\langle t_i, activity(t_i)\rangle$ through $\langle t_{i+k}, activity(t_{i+k})\rangle$ to compute the activity function. This produces one record for generating the decision tree as:

$$\langle time\ t_i, activityFunction\big(\{\langle t_j, activity\big(t_j\big)\rangle \big| where\ t_i \leq t_j \leq t_{i+k}\}\big)\rangle$$

Also, we remove the oldest activity value pair – $\langle t_i, activity(t_i)\rangle$ – and determine if $(t_{i+k} - t_{i+1}) \geq \Delta t$. If so, then we generate another record for decision tree as:

$$\langle time\ t_{i+1}, activityFunction\big(\{\langle t_j, activity\big(t_j\big)\rangle \big| where\ t_{i+1} \leq t_j \leq t_{i+k}\}\big)\rangle$$

We also continue to remove the oldest activity pairs (and to generate decision tree records) until we reach the state where the oldest activity time is t_m, and $(t_{i+k} - t_m) < \Delta t$. This ends the second cycle of the procedure, and we begin the first cycle of the procedure again, with the new starting time of t_m. We continue the procedure until we are unable to generate an interval with size greater than Δt, which also prevents us from generating any more decision tree records. In one sense, our procedure uses a producer-consumer technique, where the commodity being produced and consumed is the time interval between the oldest and most recent times in the sliding window. The expected failure duration Δt is used as a threshold value, and as a means to synchronize actions between the production and consumption cycles. The first cycle of the procedure continues until it produces an interval with a size greater than the threshold Δt, at which

point it passes control to the second cycle; similarly, the second cycle of the procedure consumes that interval to produce decision tree records until the interval size falls below the threshold Δt, at which point it passes control back to the first cycle of the procedure.

6.2.3 Clustering Technique for Determining Correlation

During the Mining and Assessment Phases, we leverage both schedule-based and demand-based relationships in our attempt to assess the operational impact. While the schedule-based relationships are based on a fixed number of time-based components (i.e. year, month, date, hour, minute and day of the week), the demand-based relationships are based on the activity values of other dependencies. Even with our initial, smaller-scale experiments, we encountered thousands of different dependencies that could be used with the demand-based relationships. Unfortunately, naively using all of the dependencies would overwhelm our system, even when using fairly powerful hardware, software and algorithms designed to handle high-dimensional data sets. Consequently, we looked for ways to reduce the number of dependencies used during demand-based relationship assessments, and to focus on those dependencies that would be more likely to yield significant results during the Mining and Assessment Phases. The algorithm used to minimize the number of relationships is given here:

Algorithm: Clustering Technique for Determining Correlation Partners

execute database **queries**
    timeStampResults() := "**select distinct** timestamp **from** usage_others
    order by timestamp **asc**";
end execute
timeIndex := 0;
initialize/empty the *timeRoster*[], *clusterCenter*[][] and *clusterGrid*[][] arrays;
for each *element* in *timeStampResults()* list **do**
    timeRoster[*timeIndex*] := *element*;
    timeIndex := *timeIndex* + 1;

```
end for
timeIncrement := timeIndex / (dimensions + 1);
timeSpan := 2 * timeIncrement + 1;
for each integer k between 0 and (dimensions – 1) do
    timeStart := k * timeIncrement;
    timeStop := timeStart + timeSpan;
    execute database queries
        delete from cluster_temp;
        insert into cluster_temp (select connection, count(*) as "frequency"
            from usage_others where timeStart ≤ timestamp and timestamp ≤ timeStop
            group by connection);
        update cluster_temp set frequency = (timeSpan – frequency)
            where frequency > (timeSpan / 2);
        frequencyResults() := select connection, frequency from cluster_temp;
    end execute
    for each record[connection, frequency] in the frequencyResults list do
        clusterGrid[k][connection] := frequency;
    end for
end for
zeroize/set to zero all empty/undefined entries in the clusterGrid[][] array;
for each link in clusterGrid[][] array do
    generate output record & store in the clusterInput file:
        ⟨link, clusterGrid[0][link], clusterGrid[1][link], …, clusterGrid[dimensions –
1][link]⟩;
end for
clusterOutput := apply WEKA clustering to clusterInput file;
for each line in the clusterOutput file do
    parse line into components: ⟨connection, clusterID⟩;
    generate output record & store in the clusterNodes file: ⟨connection, clusterID⟩;
end for
centerOutput := retrieve WEKA clustering remaining results/analysis;
for each line in the centerOutput file do
    parse line into components: ⟨clusterID, pt0, pt1, …, pt(dimensions – 1)⟩;
    for each integer k between 0 and (dimensions – 1) do
        clusterCenter[k][clusterID] := ptk;
    end for
end for
for each combination of clusterIDs (clusterP, clusterQ) where P ≠ Q do
    sumSquares := 0;
    sumNumbers := 0;
    for each integer k between 0 and (dimensions – 1) do
        if (clusterCenter[k][clusterP] = 0 or clusterCenter[k][clusterQ] = 0) then
            datapoint := 0;
        else
            datapoint := clusterCenter[k][clusterP] / clusterCenter[k][clusterQ];
        end if
```

sumNumbers := *sumNumbers* + *datapoint*;
sumSquares := *sumSquares* + *datapoint*^{2};
end for
variance := (*sumSquares* – (*sumNumbers*^{2} / *dimensions*)) / *dimensions*;
if (*variance* ≤ *varianceThreshold*) **then**
generate output record & **store** in the *clusterNodes* file: ⟨*clusterP*, *clusterQ*⟩;
end if
end for

Our goal is to determine when two dependencies – for example, dependencies *X* and *Y* – are likely to be correlated from an impact perspective. Suppose that dependencies *X* and *Y* are strongly correlated in a positive manner. Also, suppose that we are measuring the activity of *X* and *Y* over a time period from t_0 to t_n, where $activity(X, t_i) = 1$ if, and only if, dependency *X* is active at time t_i; otherwise, $activity(X, t_i) = 0$. If dependencies *X* and *Y* are strongly correlated, then $activity(X, t_i)$ should equal $activity(Y, t_i)$ in most cases over the period from t_0 to t_n. Furthermore, the sum of the activity values across the period should be fairly close, such that:

$$\left| \sum_{i=0}^{n} activity(X, t_i) - \sum_{i=0}^{n} activity(Y, t_i) \right| \leq \varepsilon(n+1)$$

where ε is an error bound/tolerance that we have selected. If the dependencies *X* and *Y* are strongly correlated over a certain period, then the sums of their activity values over that period should be fairly close. We leverage the logical complement of this statement as the basis for our technique: if the sums of the activity values are not fairly close – for example, if they are not within a certain proportional value of the size of the time period – then we propose that the dependencies are most likely not strongly correlated. We use this technique to filter out unlikely candidates for dependency testing.

We also propose refinements of this technique to handle similar cases, such as detecting negatively and positively correlated results. Suppose that, when testing over the time period from t_0 through t_n, dependency *X* is active exactly p times such that $\sum_{i=0}^{n} activity(X, t_i) = p$. If dependency *Z* has a strong negative correlation with dependency *X*, then dependency *Z* should be active approximately $(n+1) - p$ times. Consequently, when trying to determine likely correlation candidates, we should also consider those dependency pairs *X* and *Z* where:

$$\left|(n+1) - \left(\sum_{i=0}^{n} activity(X, t_i) + \sum_{i=0}^{n} activity(Z, t_i)\right)\right| \leq \varepsilon(n+1)$$

For a given set of data, we could check these dependencies in a pair-wise fashion, but this could be computationally expensive depending on the number of dependencies. We employ an alternate approach: instead of using pair-wise comparisons, we use clustering algorithms to identify groups of dependencies that have similar activity characteristics. More specifically, suppose we have activity data over a large time period from t_0 through t_n. First, we divide that period up into k smaller time periods of approximate length $m = \left\lfloor \frac{n}{k} \right\rfloor$. Though we have used equal length time periods for clarity and simplicity, there is not a requirement to use periods of equal length. Next, for each dependency, we compute the activity sum for each of the time periods, which results in a vector of the activity characteristics for that dependency. For example, the activity vector for dependency *X* would be:

$$\langle \sum_{i=0}^{m} activity(X, t_i), \sum_{i=m+1}^{2m} activity(X, t_i), \ldots, \sum_{i=(k-1)m+1}^{n} activity(X, t_i) \rangle$$

Using this vector in our clustering algorithm would detect potential candidates for positive correlation, but would not account for potential negative correlation. To handle these cases, we modify the activity by "folding" the activity values over the midpoint of the smaller time periods of length $m = \left\lfloor \frac{n}{k} \right\rfloor$. Specifically, suppose that $\sum_{i=0}^{m} activity(X, t_i) = p$, and that $\sum_{i=0}^{m} activity(Z, t_i) = (m - p)$ such that dependencies *X* and *Z* have the potential to be negatively correlated. Then, for each activity vector component that is greater than the midpoint of the range (i.e. $\frac{m}{2}$), we replace that value with the result of $m - \sum_{i=0}^{m} activity(X, t_i)$. We perform this replacement on each activity vector component of each dependency being evaluated. This transformation has the property that potentially positively correlated candidates are preserved: if the activity sum values for dependencies *X* and *Y* are very close before the transformation, then it is very likely that they are either both above, or both below, the midpoint of the range. Consequently, both activity sums will either be left unchanged, or both deducted from $\frac{n}{k}$; and, in either case, will remain close in proximity. The transformation is much like envisioning the number line of possible activity values from 0 to $\frac{n}{k}$ as drawn on a strip of paper, and then folding that paper over at the midpoint such that the ends 0 and $\frac{n}{k}$ are touching. The potentially positively correlated points are still in close proximity, and are now also in close proximity with the potentially negatively correlated points. We can now exploit these proximities by using a clustering algorithm to group dependencies according to their activity vectors, and viewing the activity vector for each dependency as a single point in an $\left\lceil \frac{n}{k} \right\rceil$ -dimensional space.

We refer to these clustering results as families, and record the dependency-to-family mapping results for later use. Also, each family contains a centroid, which is a point that best represents the "center" of the family cluster. We use the centroids as representatives for each of the families, which greatly reduces the number of comparisons needed for the later analysis, since $\|families\| \ll \|dependencies\|$. In the next step, we expand on the definition of correlation from an impact assessment standpoint. Suppose that program Q is used to process the data produced by program X, and Q is normally executed immediately after X has been completed. Because of the processing time required, user U executes program Q for 3 hours for each single hour that user U runs program X. This creates a 3-to-1 ratio in the activity sum for these dependencies (U $\rightarrow$ Q and U $\rightarrow$ X), and would generally prevent these dependencies from being detected with our current clustering process. From an impact assessment standpoint, however, there is still a relationship between X and Q: if program X is running at the time the technical event occurs, then this could affect the likelihood that program Q would be active, and potentially impacted, during the outage duration. Our intent, therefore, is to modify our clustering process to detect these relationships as well.

Our goal is to determine when two dependencies (or centroids) have a consistent ratio between their activity sums over the time periods we are measuring. One challenge is that we do not know the exact ratio: it can vary between dependencies. Consequently, if we simply measure the activity values over the entire time period, then any two dependencies will appear to have a ratio with some arbitrary value. Therefore, we divide the overall time period up into a number of smaller periods of equal length. Similar to our earlier analysis, time periods of equal length are not absolutely required, but they

make some of the calculations easier. Also, this allows us to leverage the activity sums that we computed during the initial version of the clustering process. In particular, for two dependencies X and Q, we compute the activity ratio for each corresponding component of their activity vectors as:

$$activityRatio(X, Q)$$

$$= \langle \frac{\sum_{i=0}^{m} activity(X, t_i)}{\sum_{i=0}^{m} activity(Q, t_i)}, \frac{\sum_{i=m+1}^{m} activity(X, t_i)}{\sum_{i=m+1}^{2m} activity(Q, t_i)}, \dots, \frac{\sum_{i=(k-1)m+1}^{n} activity(X, t_i)}{\sum_{i=(k-1)m+1}^{n} activity(Q, t_i)} \rangle$$

Our basic premise is that the activity ratio vector captures the activity sum ratios over a small number of fixed periods as desired. However, since we do not know a priori what value (if any) the single, "unified" ratio should have, we need a way to determine if such a single ratio really exists. If such a ratio exists, then the individual ratio values should be relatively close to the single ratio value. In fact, in an ideal case, all of the individual ratios would be equal; however, even if they are not all equivalent, they will be very close to the average of the individual values. Therefore, we view the vector components as data point in a sample, and we calculate the variance of these data points to measure how consistent and close they are to the average.

This clustering process makes our overall impact assessment processing more efficient by reducing the number of dependencies that need to be considered when assessing demand-based relationships. For a given dependency X, we can include any other specific dependencies for which we have scheduling information. In the absence of specific dependencies, we can include the dependencies that are in the same family as X. This will include dependencies that are positively and negatively correlated with X with a 1-to-1 activity ratio, such as dependencies Y and Z from our examples. Also, if we want

to consider those dependencies that are related via a different ratio, we first calculate the activity ratio vectors for the dependency X and each of the centroids for the other families. Then, for each activity vector which has a variance within our desired tolerance, we include those dependencies, like dependency Q from our example.

CHAPTER 7

EXPERIMENTAL RESULTS

7.1 Testing Small-Scale Data

We tested our approach on a computer lab with six Linux–based end-user workstations, all of which are connected to a significantly larger campus infrastructure. The collector program was implemented as a Linux batch file on each workstation, and configured to collect data at roughly 5-minute intervals, which was then consolidated to one-hour groupings. We collected data from these systems over 35 days, and then aggregated the data on a central server to support the Discovery, Mining and Assessment Phases. We gathered more than 5000 distinct groups of data from the six end-systems, distributed over approximately 700 distinct collection times. The steps taken during the Discovery, Mining and Assessment Phases allowed us to significantly reduce this potentially overwhelming amount of data, making it much more manageable and operationally relevant. There are two significant motivations in reducing the size of the system and impact topologies: to reduce the amount of information processing needed to produce an impact assessment; and, to improve the clarity of the results for the system administrators and executive users, as shown in Table 1.

The system topology data values were distributed fairly evenly around the mean. The impact topology values, however, were skewed significantly towards positive values.

Table 1 - Average Dependency Topology Sizes (Measured in Number of Dependencies/Edges)

	system-wide		***per technical event***			
	all	*real-users*	*all*	*freq < 0.1*	*0.1 ≤ freq ≤ 0.9*	*freq > 0.9*
Mean	3461	844	81	64	14	3
St. Dev.	1269	334	233	189	70	10
Skew	1.3	0.7	4.2	5.3	7.2	3.9

This was caused when certain technical events impacted an unusually large number of resources. As an example, most port or device failures only affected 4 to 12 resources. In contrast, technical events involving the *http* port on dionysos (*port* | *dionysos* | *http*), and a local device on hera (*device* | *hera* | *8-1*), impacted 405 and 1,554 resources, respectively.

The initial topology, using all of the data gathered from one collection period, has an average of 3,461 dependencies. We reduce size of the system topology by 75% by identifying the subset of this topology that has a potential impact on one or more real users. Similarly, the initial impact topology for a given technical event has an average of 81 dependencies. We reduce the number of dependencies to be evaluated for the impact assessment by 79% by eliminating those dependencies with a frequency lower than our established threshold of 10%. Finally, an average of 14 dependencies needed to be evaluated with the system usage patterns for a given technical event. We determined that 1,893 of the dependencies collected during our testing had a frequency between 10% and 90%, inclusively. Further testing showed that 1,775 of these dependencies were strongly correlated (97% or more), such that we needed to perform usage pattern mining on only 118 distinct dependencies. Our practice results so far confirm these percentages: we've had to perform usage mining on an average of 2 of the 14 dependencies, and the usage patterns for the remaining 12 dependencies were strongly correlated to these results.

We will now demonstrate these principles with a practical example. Consider the technical event caused when the *mysql* port on the six end-systems used in our test environment are closed unintentionally by a faulty host firewall configuration. The comprehensive system topology for the entire testing period included over 92,000 distinct dependencies. Manually analyzing a topology of this size would be cumbersome and error-prone. We can use automated techniques to calculate more specifically which users are likely to be affected for this event, as shown in Figure 15.

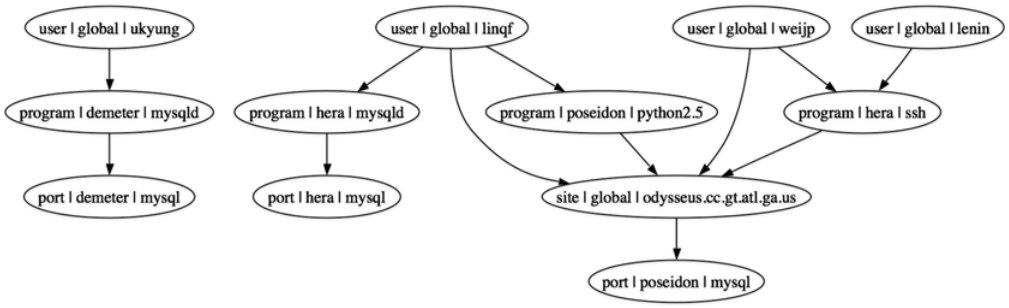

Figure 15 - Impact Topology for port | mysql Closing

Using the impact topology results alone allows us to infer that the closed *mysql* port could potentially affect 4 of the 17 total users. We can leverage the system usage patterns to more specifically determine the impact. Figure 16 gives an improved impact topology for this technical event, where each edge label represents the activity frequency for that dependency. We don't have enough information on the dependencies with a frequency < 10% to determine if they will be active during the outage period with any significant likelihood. Consequently, we remove the paths using these dependencies from consideration. The only path remaining for consideration is from *user* | *global* | *linqf* through *program* | *hera* | *mysqld* to *port* | *hera* | *mysql*.

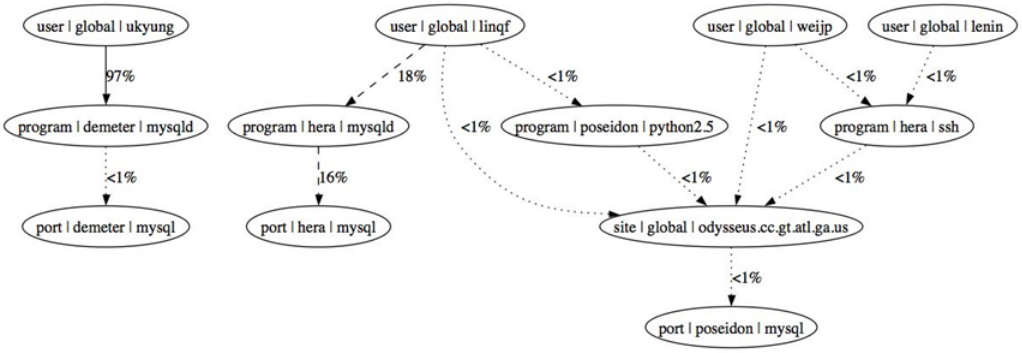

Figure 16 - Impact Topology for port | mysql Closing with Activity Frequencies

The next step is to use the timing and system status information from the technical event, along with the *system usage patterns*, to determine if there will be an impact on *user | global | linqf*. The two dependencies are strongly correlated, so we can use the same system usage pattern results for both dependencies.

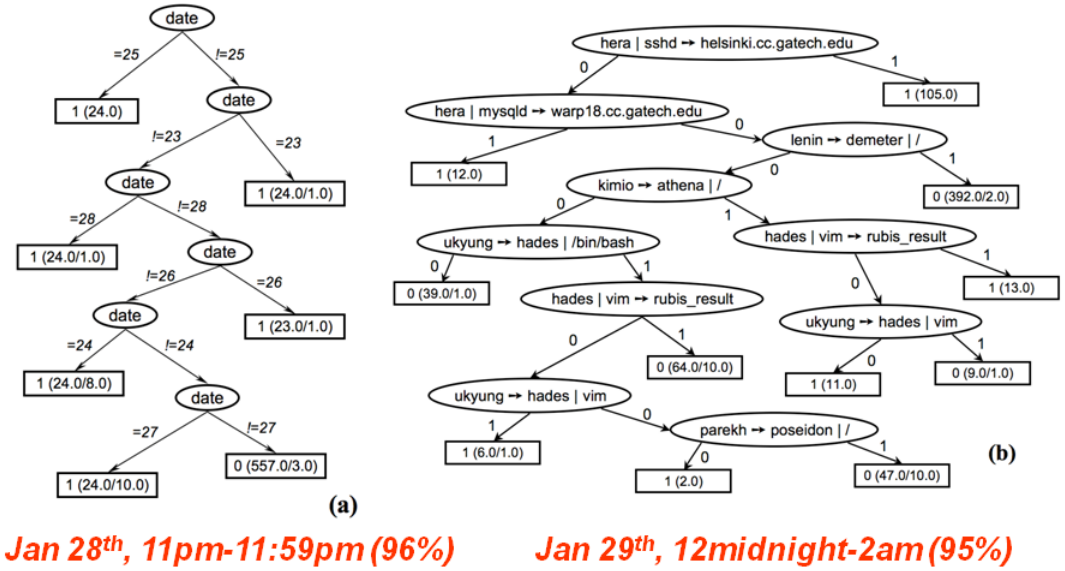

Figure 17 - Schedule- and Demand-Based Decision Trees for Usage Mining

Figure 17 shows the relevant decision tree results for these relationships. The scheduled-based decision tree has a correctly classified instances value of 96.57%, and we can use this as our measure of the likelihood of an impact. If the outage occurs between the 23rd and 28th of the month, then we would assess that user *linqf* has a 96.57% likelihood of being impacted during the outage period. Similarly, if the event occurs on the 22nd at 9pm, with an expected duration of 6 hours, then we would adjust our assessment such that user *linqf* has a 96.57% likelihood of being impacted between the hours of midnight and 3am on the 23rd.

Now, suppose the event occurs on the 15th at 4pm, and lasts 6 hours. The schedule-based patterns do not indicate activity during this period, but the demand-based patterns might still indicate activity based on the status of other resources. Our approach will assess an impact if either set of patterns – schedule-based or demand-based – indicates that the dependency is likely to be active during the outage period. The demand-based decision tree has a correctly classified instances value of 95.57%, and was generated based on the designated outage period of 6 hours. As an example, if the *sshd* program on the computer named *hera* has an active connection to the *helsinki.cc.gatech.edu* site at the time of failure, then we can infer that the dependencies *user* | *global* | *linqf* → *program* | *hera* | *mysqld* and *program* | *hera* | *mysqld* → *port* | *hera* | *mysql* will also be active at some time during the 6-hour outage period. Consequently, we would assess that user *linqf* has a 95.57% likelihood of being impacted during the outage period.

These examples demonstrate how the using the combination of system topology and system usage pattern information has allowed us to improve the clarity and

operational relevance of our impact assessments. In the given scenario, the impact topology indicates that the closed *mysql* port might impact four different users. Incorporating the usage patterns allowed us to further determine which specific users had a significant likelihood of being affected during the outage period for the failed resource.

7.2 Comparing Centralized & Distributed Processing Techniques

Given our description of these three approaches, we examine certain metrics to evaluate the tradeoffs between the different approaches. We examine the amount of data transmitted after the Collection and Discovery phases, and during the Assessment phase. We compare these results to the quality of the resulting assessments, in terms of the impacts detected and predictive strength of the resulting topologies and usage patterns. The data was collected from end-systems at the systems laboratory on Georgia Tech's campus. These machines were used over a 30-day span by various researchers employing local and system-wide applications. The data was collected at 5-minute intervals, and grouped into one-hour collection periods. We will examine each of these metrics in more detail in the following sections.

7.2.1 Data Transmission Comparisons

With the centralized approach, the raw monitoring data is sent from all end systems to the impact assessment server after the Collection phase. With the partially distributed approach, the Discovery phase is conducted at each end-system, and the global dependencies are sent to the impact assessment server, while the local dependencies are maintained at the end systems. We also considered a slight variation on the centralized approach, where the Discovery phase is conducted on the end-systems, and then all

discovered dependencies (as opposed to the raw monitoring data) are sent to the impact assessment server. The results are shown in Table 2 (file sizes given in KB).

Table 2 - Data Transmission during Collection and Discovery

	Centralized		***Partially Distributed***
	Raw Data/*Size*	Dependencies/*Size*	Dependencies (global)/*Size*
Mean	7030.3/*458.3*	230.6/*19.4*	120.9/*10.3*
St. Dev.	3033.7/*298.1*	118.2/*9.2*	68.5/*5.0*

The results show that the raw data files are many orders of magnitude larger than the comparable discovered dependency files. Even when applying the Discovery phase early in the variation on the centralization approach, the complete dependency files are still approximately twice as large as the files containing only global dependencies. From these results, it is clear that the partially distributed approach offers a significant reduction in data transmission over the centralized approach for this measurement. Also, the fully distributed approach is ideal in this case, since there is no data transmitted to the impact assessment server. Please note that the file sizes shown are for one end-system during one collection period. The total data transmitted using the centralized approach for 10,000 end systems would be approximately 4.5GB data per hour. This is not necessarily a problem for well-connected enterprises, but can cause difficulties in systems that have limited bandwidth and connectivity characteristics. Thus, the amount of data transmitted can affect the scalability of my system in certain environments.

During the Assessment phases, the data transmission rankings are reversed. The centralized approach does not require any data transmission, since all data is already located at the impact assessment server. The partially distributed approach transmits inter-zone dependencies during the Assessment phase querying process, and then

transmits the remaining affected local dependencies at the end of the process. Inter-zone dependencies involve one global component affecting (or being affected by) a local component. For the fully distributed approach, a relatively small amount of data representing the technical event information (e.g. affected component and outage duration) is sent to each end system. Then, all assessments are performed completely on each end system, and the affected dependencies are returned to the impact assessment server to assemble the final result. The results are shown in Table 3 (file sizes given in KB).

These results are partitioned according to the different types of lower-level component faults in our model. When assessing port configuration problems, the data transmission differences between the partially and fully distributed approaches are very small. The differences per assessment are more significant in the event of failed routers or devices; however, these file sizes are significantly less than those encountered during the Collection and Discovery phases.

Table 3 - Data Transmission during Assessment

	Partially Distributed	***Fully Distributed***
	Dependencies (inter-zone)/*Size*	Dependencies/*Size*
Routers		
Mean	2.6/*0.2*	5.9/*0.5*
St. Dev.	0.5/< *0.1*	1.6/*0.1*
Ports		
Mean	3.8/*0.3*	3.9/*0.3*
St.Dev.	1.9/*0.2*	2.1/*0.2*
Devices		
Mean	60.4/*5.2*	76.9/*6.6*
St. Dev.	40.7/*3.5*	70.8/*6.1*

Another key distinction is that data transmission for the Collection and Discovery phases occurs on a regular and far more frequent basis than the Assessment phase. This combination of factors indicates that we can significantly reduce the amount of data transferred with a centralized approach by using a partially or fully distributed approach instead. Of course, we must be sure that we do not significantly compromise the quality of the resulting impact assessments, which we examine the following sections.

7.2.2 Assessment Quality Comparisons

To compare the quality of the impact assessments for the different approaches, we first compared the users and top-level components affected for each infrastructure fault in my test environment. We then identified assessments with different user and top-level component results in approximately 5% and 25% of my assessments, respectively. Closer analysis of the specific instances confirmed that the differences were caused when a user accessed a specific top-level component from two or more local zones. In these cases, the user's usage frequency for that component was too small to be assessed as an impact from a local-zone perspective. The sum of the usage frequencies over all of the local zones, however, was high enough to be assessed as an impact, resulting in the assessment difference.

We also examined the system usage patterns that were derived from the data in each approach. We used the WEKA PART and J48 implementations of the C4.5 decision tree algorithm [44] to generate the usage pattern rules and statistics. Usage pattern rules are mined and extracted for each dependency. The candidate attributes for schedule-based rules include time-based values (e.g. day, month, and date). The candidate attributes for demand-based rules include the activity values for the system

dependencies. The activity status for a dependency is 1 if the dependency is detected, and 0 otherwise. The nominal attribute is the cumulative activity status for the specific dependency, which is 1 if the dependency is active at any time during the outage period.

The centralized approach mines all of the dependencies as a single group. The fully distributed approach mines each set of local zone dependencies separately. The partially distributed approach can take advantage of the global dependencies collected at the impact assessment server, as well as the local zone dependencies at each end-system. The partially distributed approach will always perform at least as well as the fully distributed approach, and possibly better since it can leverage the results at the central server. The results of these tests are shown in Table 4. We compare the correctly classified case percentages, as well as the kappa statistics as a measure of the predictive power, for each set of usage pattern rules using the different approaches.

Table 4 - Mining Quality Measurements

	Centralized		***Partially and Fully Distributed***	
	Correctly Identified Cases	Kappa Statistic	Correctly Identified Cases	Kappa Statistic
Mean	94.5	0.843	93.9	0.822
St. Dev.	2.2	0.053	2.1	0.055

The centralized correctly identified case percentage and kappa statistic values are approximately 1% and 2.5% larger than their distributed counterparts, respectively. As expected, the centralized approach has better statistics, but the difference between the approaches is not as large as we expected. We interpret this data, in combination with the assessment comparison results above, as a positive sign that we can employ distributed

impact assessment techniques to minimize data transmission without compromising the assessment results.

7.3 Testing Large–Scale Data

In our initial testing, we were able to demonstrate how are techniques assisted administrators in reducing the amount of data that they need to examine in order to assess operational impacts. Now, we demonstrate that are techniques are similarly effective on a much larger data set. The Georgia Tech network spans the campus, and includes thousands of computers, systems, services and internetworks, a significantly larger number than the six computers in our initial experiments. To collect data from the network, we leveraged the Research Network Operations Center (RNOC) CPR system. The CPR system consists of 77 computer systems co-located with routers at specific locations across the Georgia Tech network. These CPR nodes can be used to collect various types of data, and to issue commands (.e.g. ping, traceroute) as directed. We used the CPR nodes to perform distributed traceroutes, and combined that information with Netflow data from key routers to assess potential operational impacts.

7.3.1 Raw Data Collection

For our large-scale testing, we collected approximately four months of Netflow data (and associated traceroutes) from the Georgia Tech network. The data was collected between August 29th, 2008 and December 21st, 2008. As with the smaller-scale data, our primary goal is to show that the impact assessment system can assist administrators by improving their ability to make operational impact assessments. It does this by reducing the overall size of the topology to be considered based on dependency relationships, and then further

reducing the number of connections to be considered due to schedule- and demand-based timing information.

The four months of Netflow data, once processed using our lossy-counting techniques, resulted in approximately 4.1 million usage records, and over 690,000 distinct connections from a Georgia Tech system to another IP address. Our focus is on the connections originating from the CPR nodes, in order to match those connections with the routing information collected via traceroute data. Consequently, we filtered out connections originating from non-CPR nodes, such as the deploy.akamaitechnologies.com sites. This left approximately 450,000 distinct CPR-based connections.

We also collected approximately 16,000 distinct traceroutes from various CPR nodes to other sites. Since merging the Netflow usage data with the traceroute topology data is key to our impact assessment processes, we examined the intersection of the two data sets to find the connections common to both. This resulted in approximately 880,000 usage records, and 11,000 associated traceroute paths. We extracted the resulting working topology consisting of over 37,000 edges. We then decided to extract a subset of this data for further testing. We extracted the 100 most active connections, along with the associated Netflow usage and traceroute information. This subset consisted of over 54,000 usage records, and yielded a working topology with 430 distinct nodes and 550 edges. A portion of the topology is shown in Figure 18. The node contents are not intended to be easily readable; rather, the figure is intended to demonstrate the visual density of the topology as an example of the difficulty facing an administrator required to analyze this diagram manually.

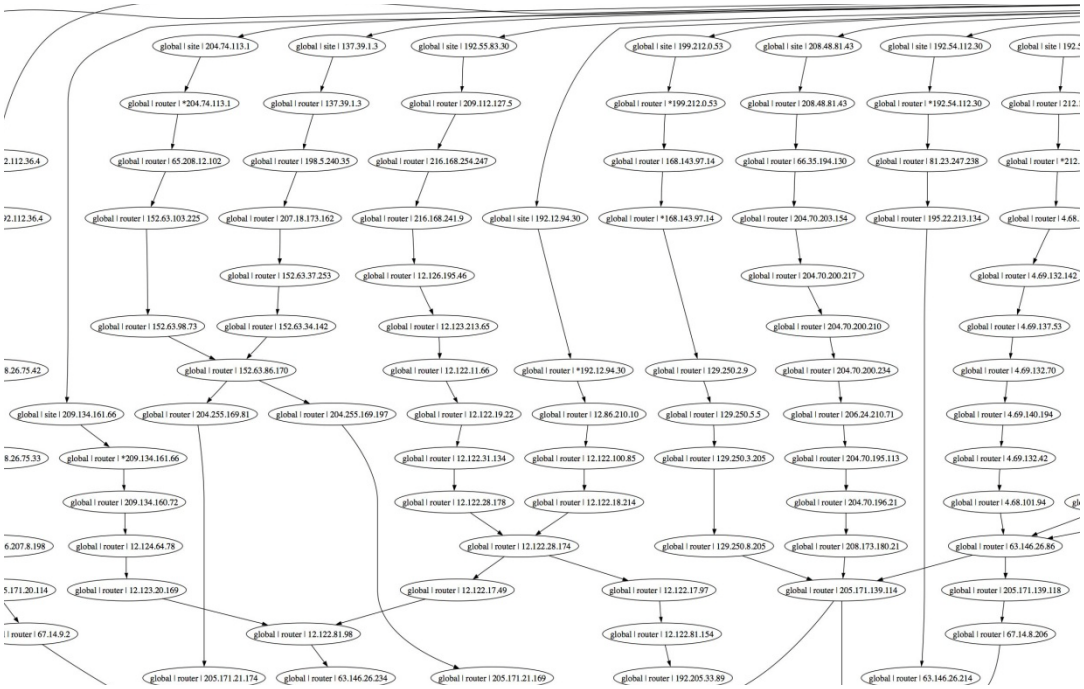

Figure 18 - Subset of the Complete Working Topology

We analyzed this data to compare the size of the resulting impact topologies compared to the working topology. We identified 261 distinct routers in the working topology, and tested each one to determine the impact for that router. From a timing perspective, we proposed failure durations of 20, 40 and 60 minutes, and that the outage would occur between Thursday, October 15th, 2009 (0001 hours) and Saturday, October 17th, 2009 (2359 hours). The time range was chosen to coincide with the Netflow data we collected; other ranges could have been used, but the accuracy of the resulting impacts might be lessened even more by the lack of relevant usage patterns for training data. The data results are given in Table 5. The data in this table represents the amount of raw connections, users and sites that were identified during one or more contiguous collection periods. The values are all measured in numbers of processed Netflow records.

Table 5 - Large-Scale Raw Data Analysis

	number of contiguous collection periods				
	1	*3*	*6*	*12*	*24*
connections					
Mean	5801.89	17405.67	34811.33	69622.67	139245.3
St. Dev.	4181.66	8734.26	13795.39	22441	41161.7
users/CPR groups					
Mean	63.61	74.58	75.42	75.83	76
St. Dev.	23.80	1.38	0.79	0.41	0
sites					
Mean	4892.49	11382.67	19562.17	33746.83	57857
St. Dev.	3567.65	5589.32	6876.77	8597.64	13993.52

Since we were working at the CPR-node level, and not identifying individual users, this limited the possible number of users/CPR nodes to 76, which is consistent with the CPR architecture at that time. In the smaller-scale testing, a complete system-wide topology had an average on 3461 edges. In our large-scale testing, a complete system-wide topology collected over one time period has an average minimum of 5801 edges (one per *user* | $U \rightarrow$ *site* | *S* dependency), not including the associated router- and origin-based dependencies that would be extracted from the traceroute data.

Given the increase in the amount of raw data being produced per collection period, our next step is to demonstrate how our impact assessment system assists in identifying a significantly smaller impact topology from the larger working topology. Figure 19 shows the distribution of impact topology sizes in terms of nodes and edges. The majority of the impact topologies had less than 30 nodes and edges. A smaller but significant number of cases range between 40 to 115 nodes, and 50 to 105 edges. Of the 261 impact test cases, 258 are shown in this figure. Three cases are not shown, and represent the most severe or "catastrophic" impact situations where a minimum of 70% of the working topology is impacted by the designated technical event. In these three

cases, the impact topologies had an average of approximately 320 distinct nodes and 390 edges. The horizontal (number of items) axis was truncated to allow better granularity for the majority of the distribution.

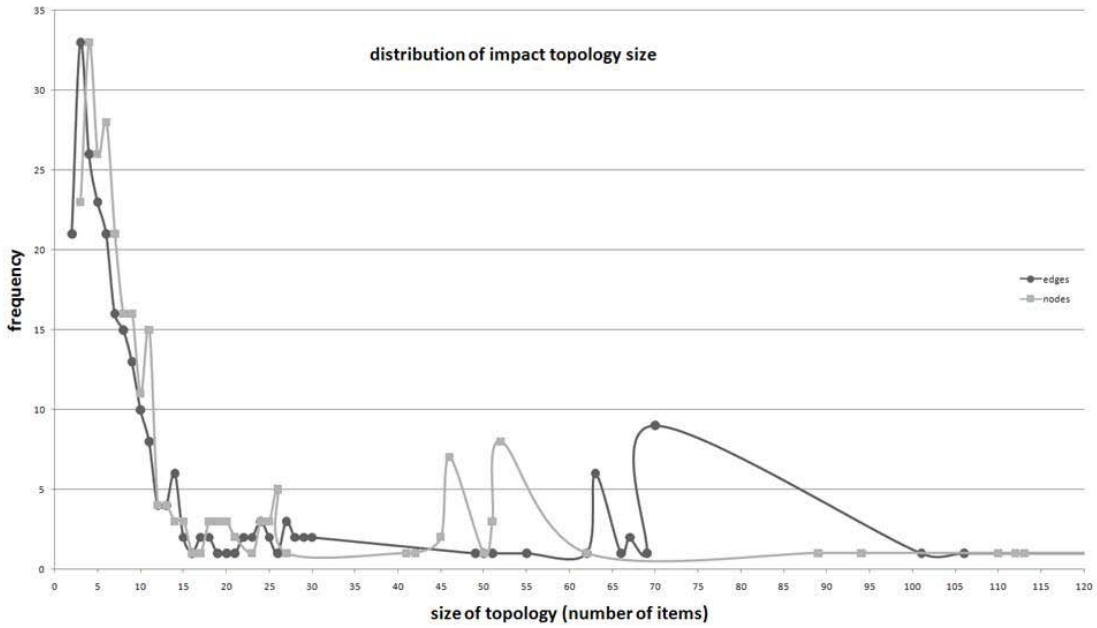

Figure 19 - Distribution of Impact Topology Sizes

Figure 20 shows the average number of users and sites impacted relative to the topology size, as measured in the number of edges. As an example, for an impact topology with 70 edges, then approximately 45 users and 55 sites would be potentially impacted by a given technical event. Note that while the average number of users and sites impacted was basically (directly) proportional to the size of the topology, it is not monotonically increasing: specifically, there are more users and sites impacted in some smaller topologies than in significantly larger topologies. For example, more users and sites are impacted on average in the topologies with 70 edges than in the topologies with 140 edges.

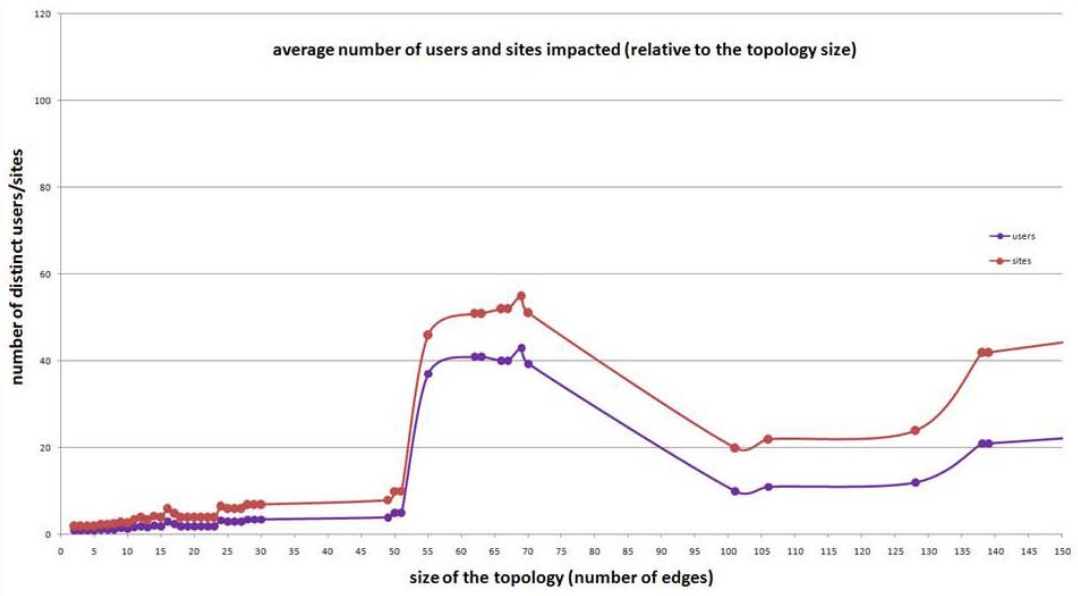

Figure 20 - Number of impacted Users and Sites (relative to Topology Size)

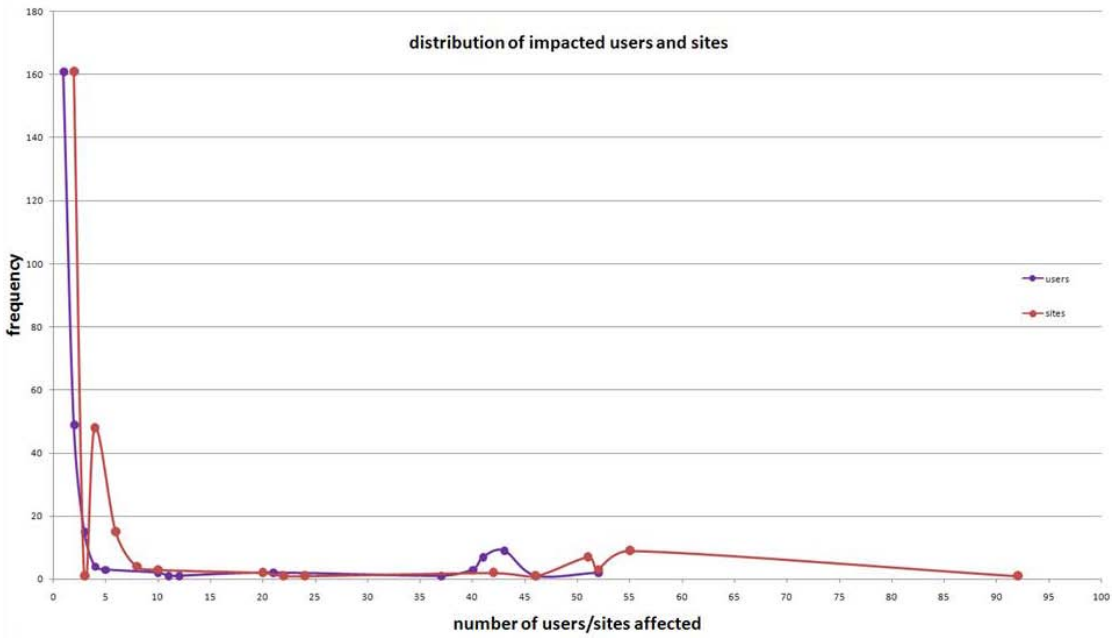

Figure 21 - Distribution of Impacted Users and Sites

Figure 21 represents the distribution of users and sites that are potentially impacted over our population of 261 impact topologies. The majority of impact topologies (approximately 200 of the 261) affected between 1 to 5 users and sites, while the remaining topologies affected between 10 to 52 users, and 6 to 92 sites.

Next, we present a summary of the size reduction analysis when considering the topology and usage/timing data in Table 6 and Table 7.

Table 6 - Topology-Based Size Reduction

	nodes	*edges*	*users*	*sites*	*routers*
mean	17.71	19.93	5.37	8.4	26.09
st dev	37.84	46.2	11.86	17.0	71.88

Table 7 - Usage/Timing-Based Size Reduction

	total impacted	*none* $f[0]$	*minimal* $f[1-5]$	*moderate* $f[6-9]$	*significant* $f[10-16]$	*severe* $f[\geq 17]$
mean	5.37	4.02	5.52%	0	15.18%	0.09%
st dev	11.86	11.37	0.29	0	0.37	0.03

As mentioned earlier, the working topology we extracted for testing purposes has 430 distinct nodes and 550 edges. We used each of the 261 distinct routers/routing points as potential failures, and then we assessed the operational impact for each failure to generate this test data. We can see that identifying the impacted connections results in an impact topology with an average size of approximately 18 distinct nodes and 20 edges, which is a significant reduction from the size of the working topology.

Furthermore, only 5.37 connections on average are potentially impacted. Of those connections, 4.02 of them will not be operationally impacted at all per the generated usage prediction model. Approximately 5– 6 of every 100 connections will be impacted minimally (with a maximum likelihood between 1% and 5%); and, 15–16 of every 100

connections will be impacted significantly (between 10% and 16%). Finally, approximately 9 of every 1000 connections will represent a potentially severe operational impact, with an impact likelihood of 17% or more. This data shows us that the size of the resulting impact assessments are generally very small compared to the overall size of the working topology, and so our system helps the administrators identify and focus on the most likely impact candidates. By the same token, the data shows that potential impacts occur enough to make monitoring this issue significant and worthwhile for many operations.

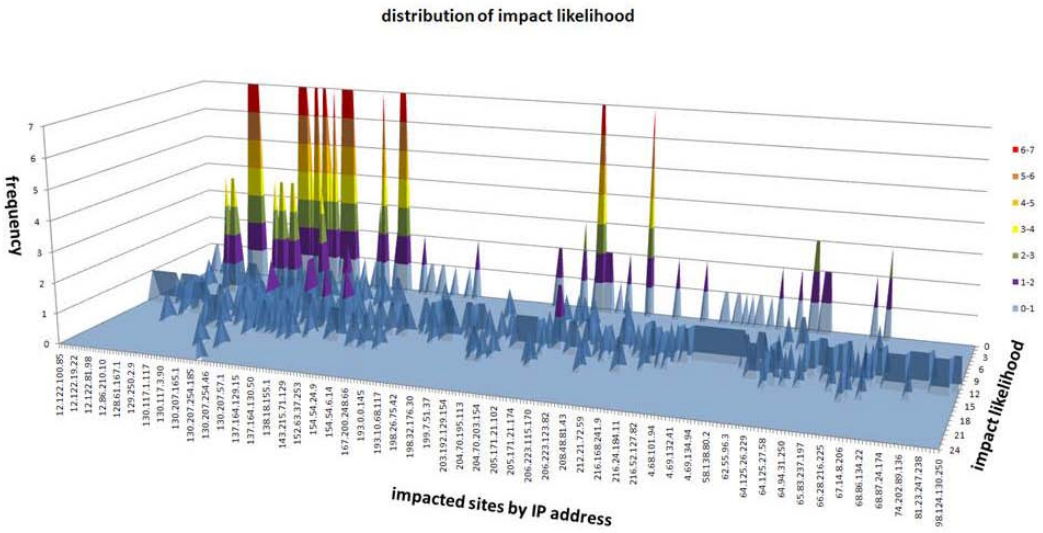

Figure 22 - Impact Likelihood Distribution across Failure Nodes

This data is represented graphically in Figure 22. This surface chart shows the distribution of operational impacts across the technical event space and range of impact likelihood values. The IP addresses along the horizontal axis represent the routers selected for simulated individual failure for the 261 technical events. Note that the impact likelihood scale (along the depth axis) is arranged in reverse order: the lower

impact values are to the rear of the chart, with the likelihood value of zero along the chart's back wall. Also, some of the $f[0]$ frequency values (along the vertical axis) for the routers between the 130.117.1.117 to 154.54.24.9 range were actually between 38 and 43, but were truncated to "7+" to improve the overall chart visibility. Note that the "ridge of peaks/mountains" running east to west along the middle of the chart floor is indicative of the 15.18% significant impact likelihood, as shown in the summary.

7.3.2 Operational Impact Assessment Examples

Next, we demonstrate a sample technical fault, and use the data we have collected to assess the operational impact on various users. The technical fault that we proposed was that the router at 143.215.194.5 would fail for 3 hours, between February 19th, 2009 and February 25th, 2009. In this instance, two connections would be affected:

- *user* | *cpr-weber* → *site* | *74.125.45.83*; and,
- *user* | *cpr-neely* → *site* | *74.125.45.83*

The impact topology is shown in the basic impact portion of Figure 23. The mitigated impact topology demonstrates that there are no alternate paths from the *origin* | *cpr-weber* and *origin* | *cpr-neely* nodes; which indicates that the impact is more likely to cause the site to be inaccessible for the user. An alternate path might lead to (at worst) an increased access time to reach the site. We also analyzed the usage patterns, and generated an impact timeline for each of these connections as shown in Figure 24. The timeline covers the period from February 19th, 2009 through February 26th, 2009. While the usage patterns for both connections are fairly consistent, there is a clear increase in activity for both connections on February 24th.

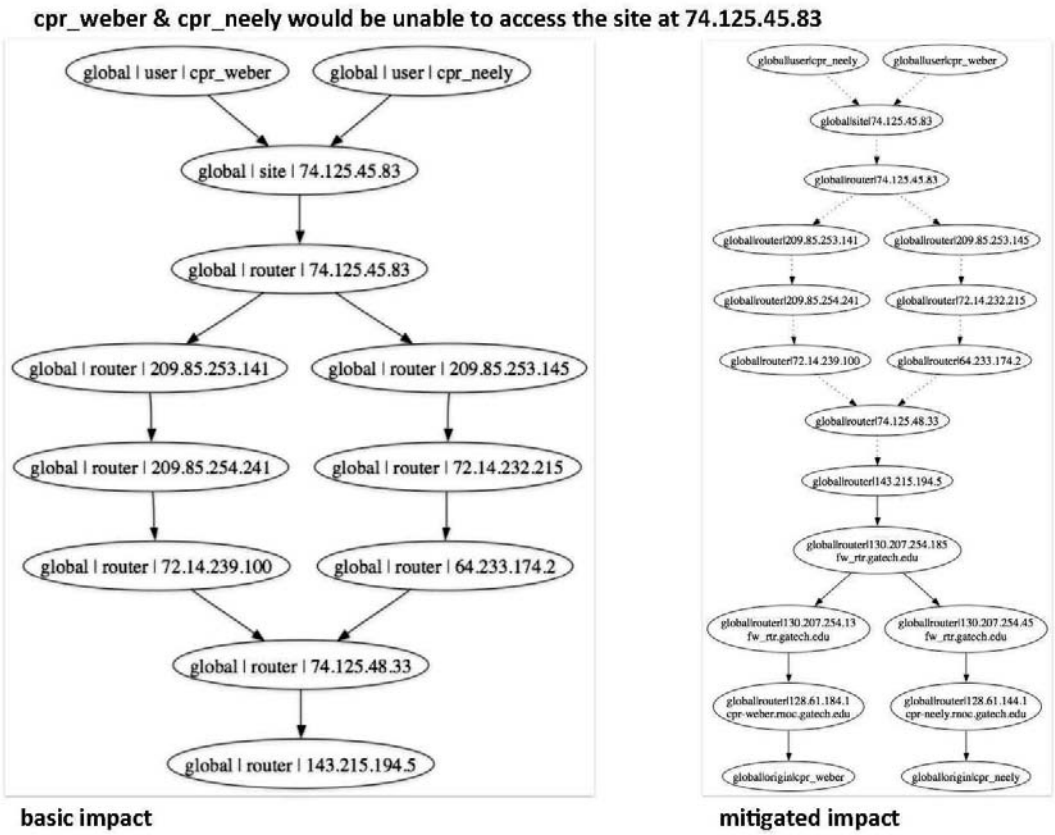

Figure 23 - Basic and Mitigated Impact for Router Failure

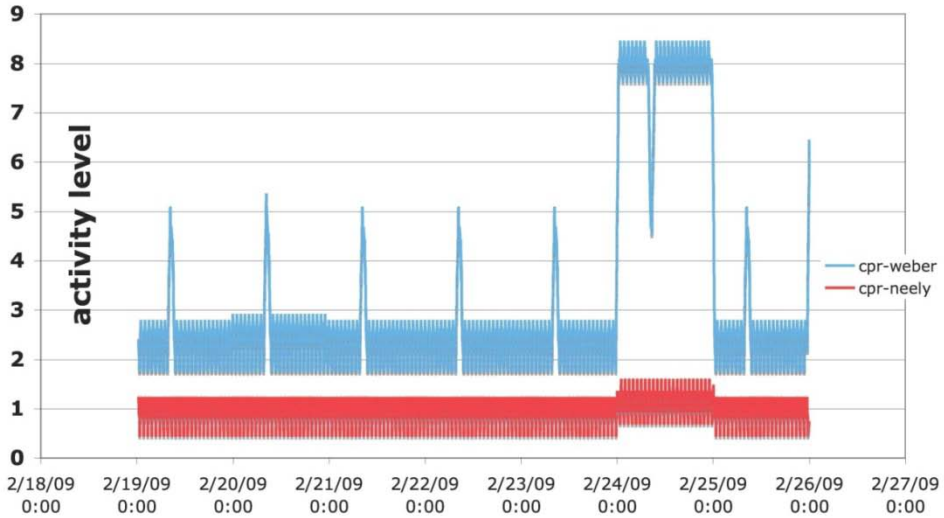

Figure 24 - cpr-weber and cpr-neely Activity During 19-26 Feb 2009 Period

This could be used by an administrator as an indicator that the router failure might have had a much more significant operational impact on the weber and neely users during that period than if it had failed on one of the other days. Also, the increase in impact could be used as guidance to avoid actions that could adversely affect the router's performance (e.g. maintenance requiring downtime) during that period, if our system is being used as a forecasting tool.

Similarly, we adjusted the timeline to determine those periods of increased and decreased operational impact for cpr-neely and cpr-weber over the March 15, 2009 through March 31, 2009 period, as shown in Figure 25.

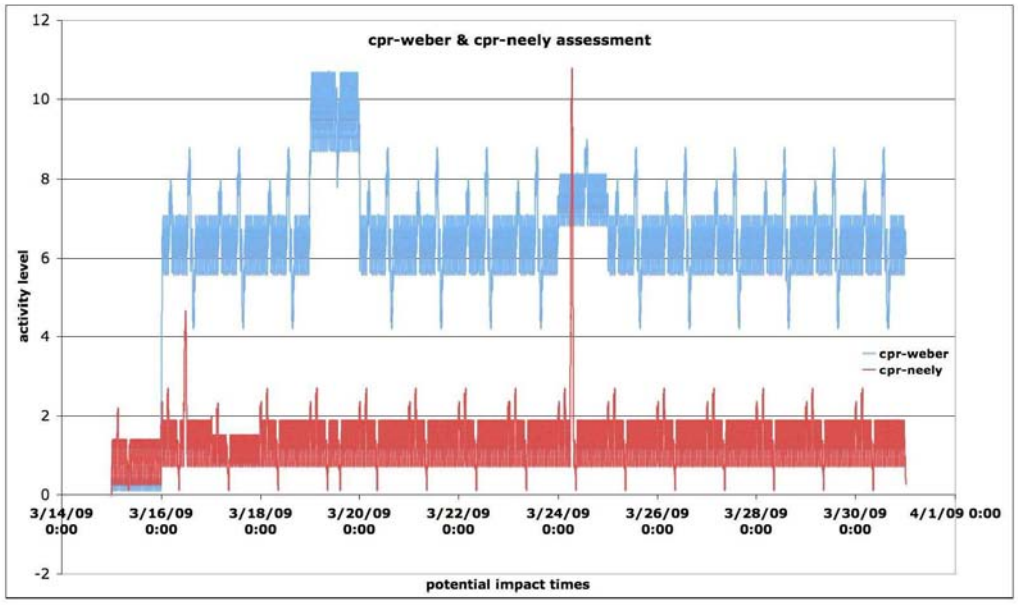

Figure 25 - cpr-weber and cpr-neely Activity During 15-31 Mar 2009 Period

The graph indicates that March 15th and March 17th are periods when the potential operational impact, especially for cpr-neely, would be significantly lower than normal levels. In contrast, the periods of March 19th and March 24th indicate a significantly higher operational impact likelihood, especially for cpr-weber. The intent of our tool is

to provide administrators with these kinds of visual graphs, so that they can not only determine specific periods of increased and decreased operational impact, but so they can also see the patterns of usage over time.

We also used our impact assessment system to forecast and compare the operational impact for a different problem. Figure 26 shows a subset of the working topology for the connections that would be affected by the failure of the North Interconnect router. We extracted six of the most active connections in terms of the number of usage records contained in the Netflow dataset.

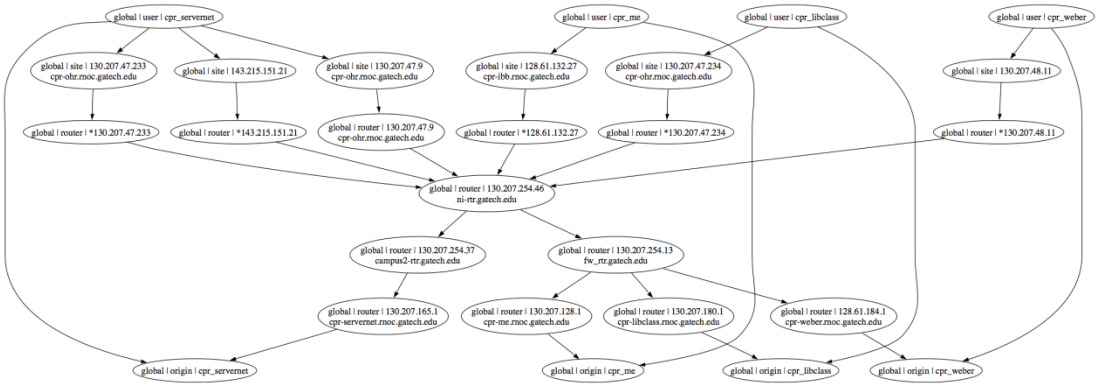

Figure 26 - Working Topology for North Interconnect related Connections

We then generated the operational impact timelines for these connections over the March 5, 2009 to March 8, 2009 time period. We then constructed a "combined" impact timeline by displaying the maximum, average and minimum values for the set of connections, as shown in Figure 27. We also identified a subset of the working topology that included connections with a IP address related to either of the international Georgia Tech campuses in France or Shanghai, as shown in Figure 28.

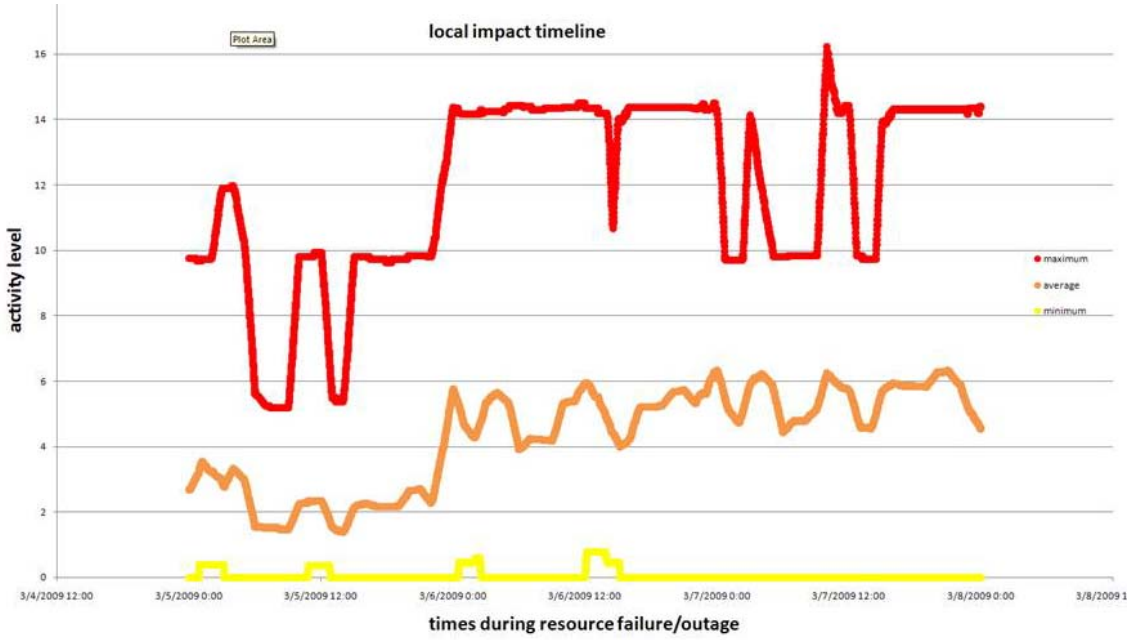

Figure 27 - Impact Timeline for Local Connections

We focused on connections that had a reasonably significant number of usage records from our Netflow data set, in order to increase the probability that we would have enough data to generate a timeline. Note that most of the ten connections are from the *cpr-servernet* node, though there are also two connections from the *cpr-me* and *cpr-nt* nodes as well (more clearly visible in the lower half of the topology). Upon further analysis, the individual usage timelines for three of the connections indicated no impact for the entire time period; therefore, we eliminated the connections from further consideration. We then generated the combined timeline for these remaining seven connections, as shown in Figure 29.

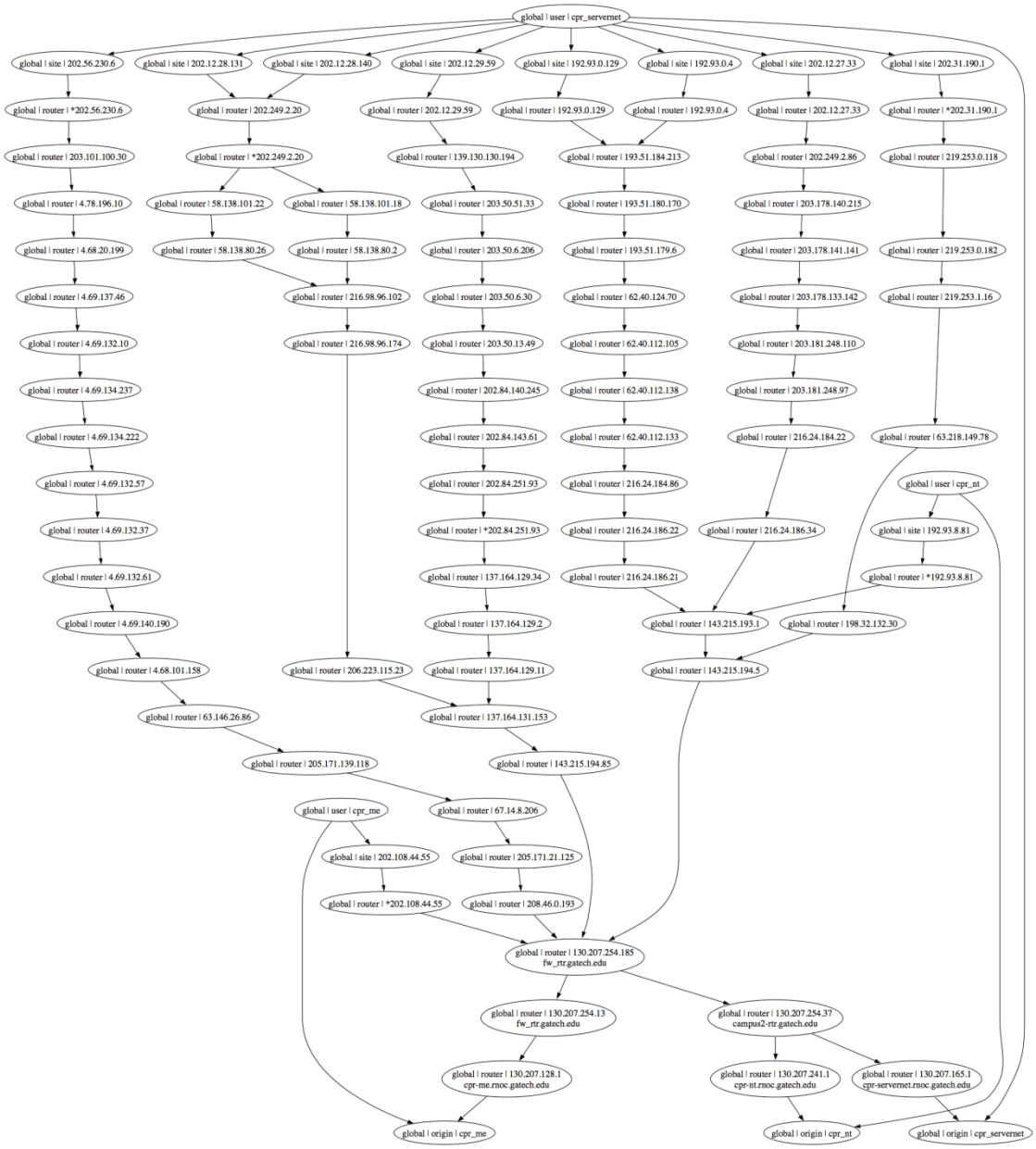

Figure 28 - French and Shanghai-related GT connections

Finally, we combined the local and global timelines to produce a combined, worldwide timeline in Figure 30. Observe that the combined timeline format offers some flexibility in determining the optimal times for minimizing operational impact. Our initial instinct was to identify the most likely periods for minimizing impact by looking

for the lowest points in the maximum curve in the timeline, giving secondary emphasis to the average and minimum curves as required. Since the maximum curve represents the highest impact value over the set of connections, we know that at least one connection holds that level of activity at that point in time. However, all of the other connections might hold very low values during that same period, such that the true system-wide impact might still be fairly low. By the same reasoning, the minimum curve assures us that all of the connections hold at least that level of activity at that point in time.

Consequently, when there is a significantly increase in the minimum curve level (for example, in the global impact timeline between March 5th and March 6th), then we are more assured of an increased system-wide operational impact, as opposed to the potential increase for only one connection. Furthermore, the average curve can be used to better predict the distribution of the entire set of connections being assessed.

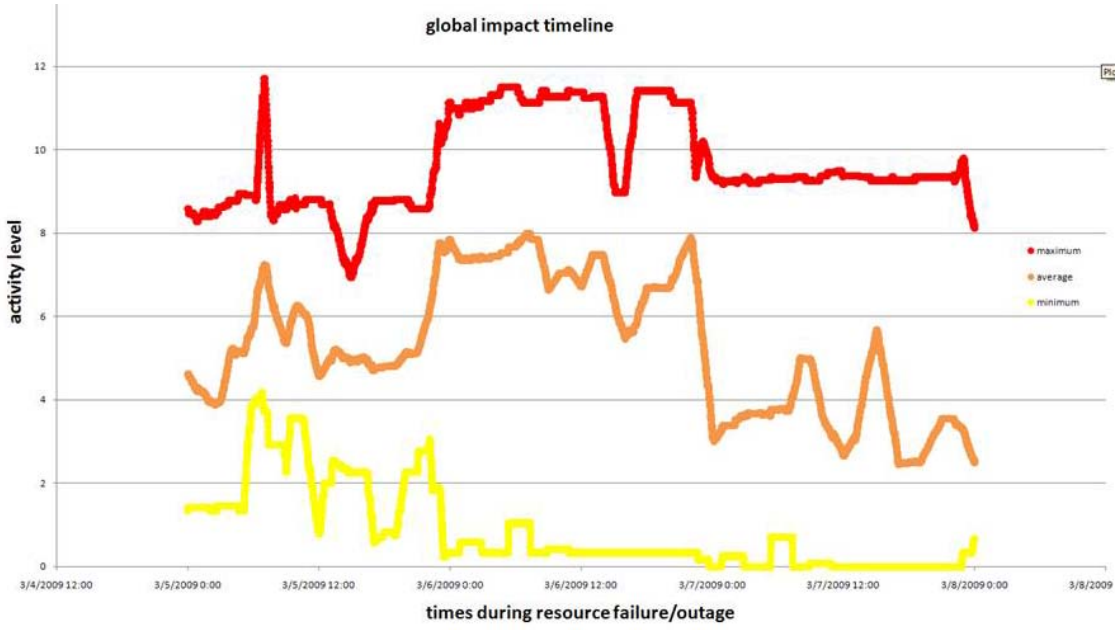

Figure 29 - Impact Timeline for Global Connections

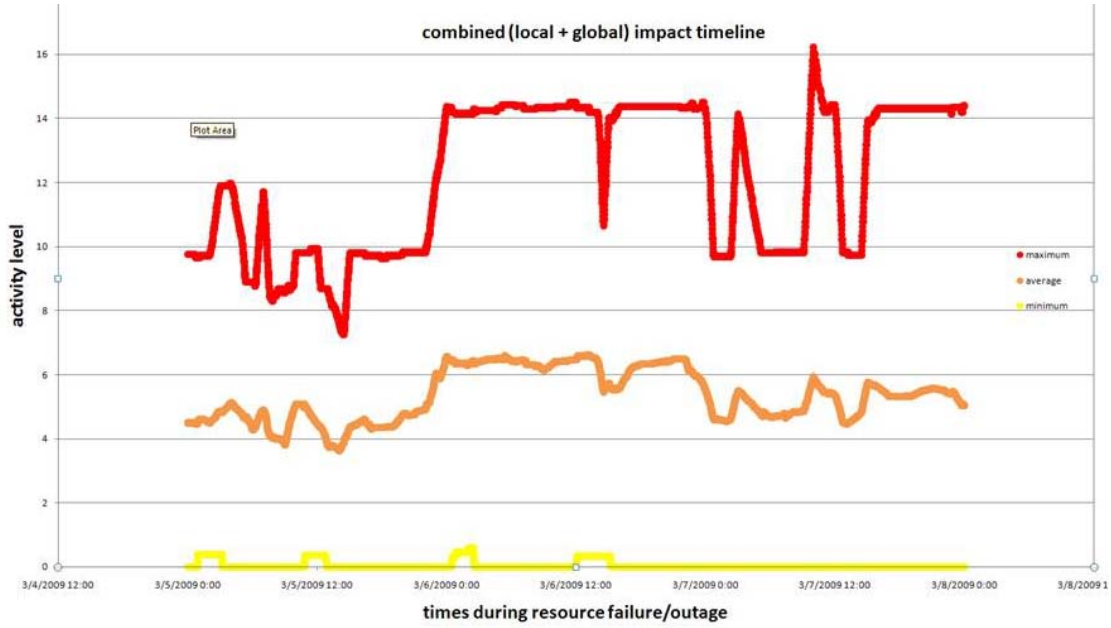

Figure 30 - Impact Topology for Worldwide Connections

As stated earlier, it is our intent to provide administrators with tools to help them assess and minimize the operational impact on their systems. We believe that the impact topologies and combined timeline(s) produced by our operational impact assessment systems offer ways to better visualize, quantify and assess these kinds of impacts.

7.3.3 Clustering Effectiveness

The clustering techniques that we described earlier are intended to reduce the number of comparisons required when assessing demand-based dependencies. The basic principle is to identify dependency groupings that have a reasonable likelihood of being correlated; similarly, eliminating those pairs that are unlikely to be correlated. We do not have to explicitly test for correlation – in fact, once the groupings are identified, we include the

dependencies from a given group as input for the assess_timeline() and assess_frequencies() procedures.

We used our working topology with the top 100 most active connections. We compare three different clustering functions: activity-based version with positive and negative correlation testing, as described earlier; an activity-based version with only positive correlation grouping; and, a frequency-based version with positive correlation grouping. The frequency-based version is similar in concept to the activity-based version, but it calculates the average of the frequency values during each time interval, as opposed to calculating the sum of the binary activity values. Also, because of the functional differences, we do not attempt to apply the same "folding" technique to detect negative correlations with the frequency-based version. Finally, we vary the number of dimensions used in the testing, which corresponds to the number of time intervals that are used to sample the activity values (or frequencies) for each connection. The results of the clustering testing are shown in Table 8.

Table 8 - Activity- and Frequency-Based Clustering Analysis

	dimensions							
Version	*2*	*3*	*4*	*5*	*10*	*15*	*20*	*25*
activity-based *positive + negative*	4/32.16	4/41.3	2/82.85	3/58.64	2/64.58	3/64.74	3/62.06	4/43.58
activity-based *positive only*	4/32.16	4/41.3	2/82.85	3/58.64	2/64.58	3/64.74	3/81.68	4/43.58
frequency-based	3/57.74	4/39.14	4/35.78	6/21.5	4/49.18	4/48.86	4/52.46	4/49.44

The results in the table use the format:

$\langle number\ of\ clusters \rangle / \langle weighted\ average\ cluster\ size \rangle$

The weighted average cluster size represents the average cluster size that is expected if we perform demand-based assessments using the clustering results. For example, suppose we have partitioned the total number of N connections over k distinct clusters. Then, if connection *c* is selected for assessment, and *c* belongs to $cluster_m$, then we would also include the rest of the connections in $cluster_m$ for assessment as well. Consequently, we compute the weighted average cluster size as:

$$weightedAverageClusterSize = \frac{1}{N}\sum_{i=1}^{k} |cluster_i|^2$$

The weighted average group size makes the assumption that each of the N connections has an equal probability for being selected for an assessment. We could possibly remove this assumption of equality, and calculate a probability distribution for the different connections based on the working and impact topologies along with other factors; however, I feel that the equality assumption is reasonable at this stage of our investigation.

The actual results were somewhat surprising. We did not expect such similar results between the activity-based versions – we felt that including the folding technique for detecting negative correlations would make more of a difference. The results show that the two versions produced identical results except for the 20-dimensions case. Similarly, we did not expect the difference between the frequency-based and activity-based versions to be so dramatic in favor of the former. The frequency-based version yields smaller and more evenly distributed clusters for all of the cases except the extremes of 3 and 25 dimensions.

7.3.4 System Performance Testing

An important goal is to ensure that our impact assessment system can actually produce assessments in a reasonable amount of time. This means that our system needs to process the incoming data at a rate equal to, or faster than, the data is being received. When a technical event occurs, our system needs to produce an assessment in minutes. If it takes hours (or longer) to produce an assessment, then the results might not be available in time for the administrators and/or executives to make a timely decision. We were conscious of this requirement throughout the development of our system.

The RNOC CPR data gave us the most realistic, large-scale tests, so we analyzed that data for our performance tests. The RNOC CPR files were initially processed remotely on IBM Blade Servers with Quad Xeon processors and 1 GB RAM. The CPR files are generated in the Cisco Netflow format. The first step of our processing requires that we use the Cisco flow-export tool to translate the raw data files from the Netflow format into a more manageable format. The CPR files contained an average of 3,100,000 lines per file, where each line represents a connection from a source to a destination at one specific point in time. It takes an average of 37 seconds to extract the relevant source, destination and timing fields from each line in the raw data file, and to convert the source and destination fields from a binary to an IP address format. The resulting "intermediate format" CPR files were then sent to the local system for further processing.

We performed the remaining testing on a local Apple MacBook Pro running Mac OS X version 10.5.6, with a 2.4 GHz Intel Core 2 Duo processor and 4 GB of RAM. Each intermediate CPR file represents 5-minutes of real-time data. Our system processes one of the intermediate files by filtering the most frequent connections using our variant

of the lossy-counting algorithm. Our system can process one of these files in an average time of 1 minute and 35 seconds. This is more than sufficient for our requirements: upon receiving a single CPR file, our system can process the raw data using the flow-export tool, filter the most frequent connections, and then append the new information to the core database tables in approximately 2 minutes and 12 seconds, leaving a margin of 2 minutes and 48 seconds before the next CPR file is generated.

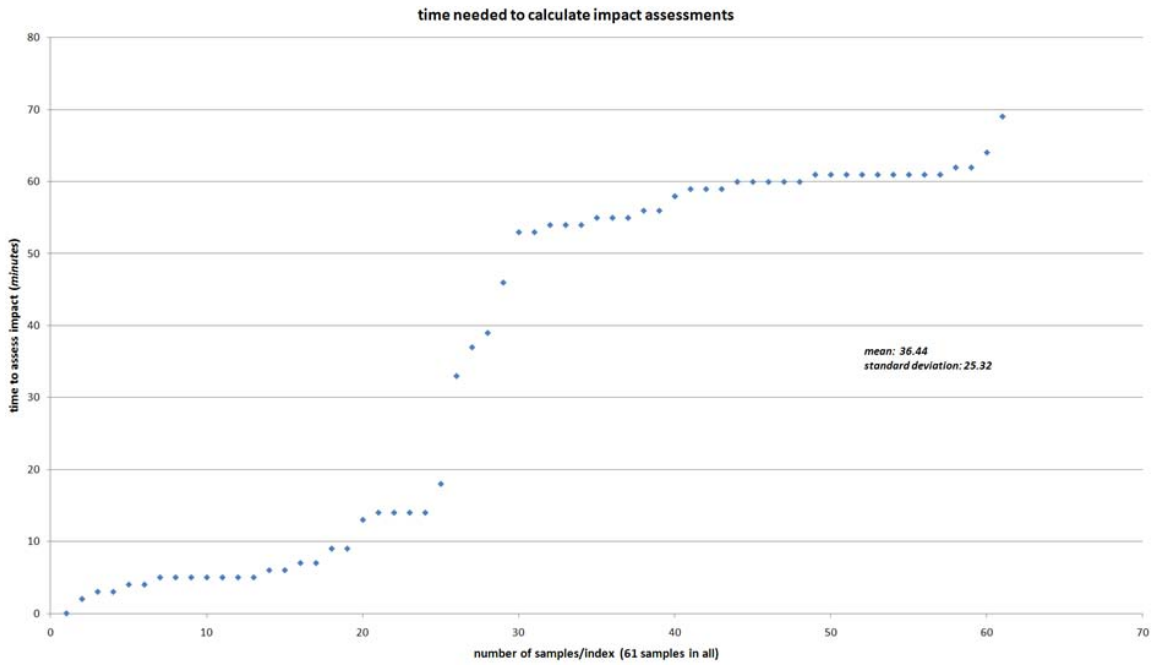

Figure 31 - Time Required for Impact Assessments

We also tested a number of scenarios in which we generated sample operational impact assessments based on a randomly generated technical event. These assessment tests were also generated on the local MacBook Pro system described above. We sampled 61 technical events we generated for our earlier testing were timed. The earlier testing gives the sizes of the working and impact topologies. Based on these topology

sizes, our system is able to generate an impact topology in an average period of 37 minutes, with a standard deviation of 25 minutes – the distribution is shown above in Figure 31. The time needed to generate an impact assessment varies directly with the number of user-resource connections that are affected by the technical event, since each connection must then be mined against the event timeline to determine the impact likelihood. Even in the worst of our sampled cases, our system was able to generate an assessment in 69 minutes, which should provide enough time for administrators and executives to direct operational changes based on the outcome of the assessment.

CHAPTER 8

CONCLUSION & FUTURE RESEARCH

We have described the framework and dataflow architecture for an operational impact assessment model and system that integrates events from all system and application components. By clustering events through simple data mining and statistical techniques, our system translates a low level event (e.g., failure of a device or router) into an operational impact assessment meaningful to system administrators and managers. We implemented our system as a working prototype, and used it to conduct tests on smaller-scale and large-scale data. We also demonstrated distributed assessment techniques designed to minimize the resources of the systems (e.g. network bandwidth) on which our operational impact system is being implemented. Our results confirmed that the distributed versions can produce impact assessment results comparable in quality to the centralized version, while significantly reducing the amount of data transferred across the network. We tested our approach on a smaller scale by collecting and analyzing operational data at the Georgia Tech Center for Experimental Research in Computer Systems (CERCS) Laboratory over a 35 day period. We have also conducted similar tests on large-scale data collected from Georgia Tech's campus network over a four month period. Our experimental results have shown that our operational impact system, procedures and techniques can assist administrators by assisting them in identifying the actual impact topologies, and by leveraging the usage data to predict if the resources that are being assessed would actually be in use during the technical event period (e.g. unexpected failure, planned maintenance outage).

Considering all that has been done so far, there are still many possibilities for future research. From the system management point of view, we consider the work described here as a solid step towards similar efforts. We should consider ways to test the effectiveness of the system on real-world impacts: for example, actually using the system to plan an event that could have an operational impact on the users, and then measuring and evaluating the results (including user feedback) as accurately as possible to determine the systems effectiveness. Our goal has been to show the potential of the system to assist by reducing the managing the size and complexity of assessing operational impact in a complex environment, and providing tools which administrators, and executives without an intensive IT background, can use to better understand and visualize the potential impact that technical events like component failures can have on their actual operations – business, military, or otherwise. The potential has been demonstrated, but there is still more testing required if we are to quantify how accurate our assessments actually are in practice.

Another possibility is to continue to develop and refine the user interface. The system we have developed is much more complete and comprehensive than our original system, which still required a significant level of manual interaction, including transferring large amounts of data between different files and databases. The current system provides a more unified structure for storing, transforming and analyzing data as required. Still, the current user interface is command-line driven. While this has some advantages (e.g. scripting certain tasks), ideally we would like to see a web-based/graphical interface developed to allow administrators to use the tools more easily. As one example, the user should be able to view, select, zoom in/out, and filter out

different sections of a large topology easily. Also, we should continue to develop ways to visualize the impact timeline as well, and to perhaps integrate the impact topology and timeline into a unified view. There may even be some promise in viewing the topology like a weather map, where periods of significant or severe impact would be displayed much like a storm moving across a geographical area; for example, using reds and oranges to denote periods of adverse operational impact, and blues and greens to denote minimal or no impact predicted.

Also, the system structure itself should be developed further. Two examples of future improvements include strengthening the database for centralized operations, and designing ways to make the key procedures more efficient and scalable for large data sets. We are constantly using the Derby Java-based database for our system. This choice was based primarily on our efforts to make the system portable in order to better support distributed operations as needed. The Derby database has worked well, especially during this prototyping and developmental phase of the research. As the system matures, however, we should investigate other database systems that can perform well with larger data sets. This is especially important considering our latest research efforts: in working with Georgia Tech's RNOC Group, we have been processing significantly larger data sets in a centralized fashion. Our system has been designed and implemented to allow a different database to be substituted if required – the queries we used to interact with the database has been written using basic SQL, and we have avoid proprietary features and extensions as much as possible. Some possibilities include MySQL to retain the portability option, or possibly an Oracle, Sybase or SQL Server enterprise-level database for more dedicated centralized analysis.

On improving the key algorithms, we should investigate ways to streamline the processes now that we have better defined the dataflow between components, along with the data structures for storage. One example is that we are using the WEKA data mining suite to perform a number of different tasks, such as generating decision trees and the equivalent rule sets for assessing the impact timelines. The WEKA suite has been wonderfully powerful and flexible, especially in investigating different methodologies during our initial design phases. Now, however, we believe that we should also invest time investigating ways to increase the efficiency of the processes to support quick and efficient analysis in real-world environments. There are alternative data structures that could be used, such as Concept-adapting Very Fast Decision Trees (CVFDTs), which might prove ideal for our intentions [39].

REFERENCES

[1] F. Mamaghani, "Impact of Information Technology on the Workforce of the Future: Analysis," in *International Journal of Management*, vol. 23, number 4, December 2006, p. 845.

[2] B. Stone, "As Web Traffic Grows, Crashes Take Bigger Toll", New York Times, July 6, 2008, http://www.nytimes.com/2008/07/06/technology/06outage.html (date accessed: 20 May 2009)

[3] M. Gramaila, R. Mills, L. Fortson. Improving the Cyber Incident Mission Impact Assessment (CIMIA) Process. In *Proceedings of the 4th Annual Workshop on Cyber Security and Information Intelligence Research (CSIIRW '08)*, article 32, Oak Ridge, Tennessee, May 2008.

[4] Defense Information Systems Agency (DISA) Host-Based Security System (HBSS) Fact Sheet (date accessed: 20 May 2009) http://www.disa.mil/news/pressresources/factsheets/hbss.html

[5] A. Friedman. "A Way to Operationalize the DoD's Critical Infrastructure Protection Program Using Information Assurance Policies and Technologies", U.S. Army War College Strategy Research Project, Carlisle Barracks, PA, March 2005.

[6] M.-C. Shan, "Intelligent business operation management," in *Proc. 2005 IEEE International Conf. on Granular Computing*, vol. 1, July 2005, pp. 14-15.

[7] C. Pu, M. Moss. Assessing Operational Impact in Enterprise Systems by Mining Usage Patterns. In *Proceedings of the 18th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 2007)*, pages 159-170, Springer, San Jose, CA, October 2007.

[8] M. Moss. Comparing Centralized and Distributed Approaches for Operational Impact Analysis in Enterprise Systems. In *Proceedings of the 2007 IEEE International Conference on Granular Computing*, pages 765-769, IEEE Computer Society, San Jose, CA, November 2007.

[9] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds and A. Muthitacharoen. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 74-89, Bolton Landing, NY, October 2003.

[10] Department of Defense Instruction (DoDI) 8580.1, *Information Assurance (IA) in the Defense Acquisition System*. Assistant Secretary of Defense, National Information Infrastructure (ASD-NII), July 9, 2004.

[11] A. Hanemann, D. Schmitz and M. Sailer. A Framework for Failure Impact Analysis and Recovery with Respect to Service Level Agreements. In *Proceedings of the IEEE International Conference on Services Computing (SCC)*, pages 49-56, volume 2, July 2005.

[12] D. Jobst, G. Preissler, Mapping Clouds of SOA- and Business-related Events for an Enterprise Cockpit in a Java-based Environment, In *Proceedings of the 4th International Symposium on Principles and Practice of Programming in Java*, pages 230-236, volume 178, Mannheim, Germany, 2006.

[13] E. Thereska, D. Narayanan and G. Ganger. Towards self-predicting systems: What if you could ask "what-if"? In *Proceedings of the Sixteenth International Workshop on Database and Expert Systems Applications*, pages 196-200, Copenhagen, Denmark, August 2005.

[14] A. Singh, M. Koropolu and K. Voruganti. Zodiac: Efficient Impact Analysis for Storage Area Networks. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, San Francisco, California, December 2005.

[15] S. Hariri, G. Qu, T. Dharmagadda, M. Ramkishore and C. Raghavendra. Impact Analysis of Faults and Attacks in Large-Scale Networks. IEEE *Security & Privacy Magazine*, pages 49-54, September-October 2003.

[16] M.-A. Jashki, R. Zafarani, E. Bagheri. Towards a more efficient static software change impact analysis method. In *Proceedings of the 8th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '08)*, pages 84-90, Atlanta, GA, November 2008.

[17] R. Walker, R. Holmes, I. Hedgeland, P. Kapur, A. Smith. A lightweight approach to technical risk estimation via probabilistic impact analysis. In *Proceedings of the 2006 International Workshop on Mining Software Repositories (MSR '06)*, pages 98-104, Shanghai, China, May 2006.

[18] A. Keller, U. Blumenthal and G. Kar. Classification and Computation of Dependencies for Distributed Management. In *Proceeedings of the Fifth IEEE Symposium on Computers and Communications*, pages 78-83, Antibes-Juan les Pins, France, July 2000.

[19] G. Kar, S. Keller and S. Calo. Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis. *IEEE/IFIP Network Management and Operations Symposium (NOMS)*, pages 61-74, Honolulu, HI, April 2000.

[20] A. Brown, G. Kar and A. Keller. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. *IEEE/IFIP International Symposium on Integrated Network Management*, pages 377-390, Seattle, WA, May 2001.

[21] C. Ensel. A Scalable Approach to Automated Service Dependency Modeling in Heterogeneous Environments. In *Proceedings of the Fifth IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, pages 128-139, Seattle, WA, September 2001.

[22] M. Chen, E. Kiciman, A. Accardi, A. Fox and E. Brewer. Using Runtime Paths for Macro Analysis. *9th Workshop on Hot Topics in Operating Systems*, Lihue, HI, May 2003.

[23] M. Chen, E. Kiciman, E. Fratkin, A. Fox and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 595-604, Washington, DC, June 2002.

[24] E. Kiciman and A. Fox. Detecting Application-Level Failures in Component-Based Internet Services. In *IEEE Transactions on Neural Networks*, pages 1027-1041, volume 16, issue 5, September 2005.

[25] J. Srivastava, R. Cooley, M. Deshpande and P.-N. Tan. Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data. *SIGKDD Explorations*, pages 12–23, volume 1, issue 2, January 2000.

[26] W.M.P. van der Aalst, B.F. van Dongen, J. Herbst, L. Maruster, G. Schimm and A.J.M.M. Weitjers. Workflow Mining: A Survey of Issues and Approaches. *ACM Data & Knowledge Engineering*, pages 237-267, volume 47, issue 2, November 2003.

[27] EMC^2|SMARTS Business Impact Manager (date accessed: 9 April 2009); available from http://www.emc.com/products/software/smarts/bim/

[28] IBM Tivoli Application Dependency Discovery Manager (date accessed: 9 April 2009); available from http://www-306.ibm.com/software/tivoli/products/taddm/

[29] J. Stanley, R. Mills, R. Raines, R. Baldwin. Correlating Network Services with Operational Mission Impact. In *Proceedings of the IEEE Military Communications Conference (MILCOM 2005)*, pages 162-168, volume 1, October 2005.

[30] R. Mahajan, N. Spring, D. Wetherall and T. Anderson. User-level Internet Path Diagnosis. *Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 106-119, Bolton Landing, NY, 2003.

[31] S. Sitaraman and S. Venkatesan. Forensic Analysis of File System Intrusions using Improved Backtracking. In *Proceedings of the Third IEEE International Workshop on Information Assurance (IWIA)*, pages 154-163, Volume 00, College Park, MD, March 2005.

[32] S. King and P. Chen. Backtracking intrusions. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 223-236, Bolton Landing, NY, October 2003.

[33] J.-M. Guillaume and M. Latapy. Relevance of Massively Distributed Explorations of the Internet Topology: Simulation Results. In *Proceeedings of the 24th Annual Joint Conference of the IEEE Computer and Communications Societies*, pages 1084-1094, volume 2, March 2005.

[34] G. Connolly, A. Saschenko and G. Markowsky. Distributed Traceroute Approach to Geographically Locating IP Devices. *Proceedings of the Second IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, pages 128-131, September 2003.

[35] W. Liu and R. Boutaba. Tracerouting Peer-to-Peer Networks. *Workshop on End-to-End Monitoring Techniques and Services (E2EMON)*, pages 101-114, May 2005.

[36] R. Mortier, R. Isaacs and P. Barham. Anemone: using end-systems as a rich network management platform. *Microsoft Technical Report, MSR-TR-2005-62*, Microsoft Research, Cambridge, UK, May 2005.

[37] W. Stevens and S. Rago. *Advanced Programming in the UNIX Environment*, Second Edition. Addison-Wesley Publishing, 2005.

[38] Graphviz – Graph Visualization Software (date accessed: 21 May 2009) http://www.graphviz.org/

[39] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*, Second Edition. Morgan-Kaufmann Publishers, 2006.

[40] C. Tang, R. Chang, E. So. A Distributed Management Infrastructure for Enterprise Data Centers Based on Peer-To-Peer Technology. In *Proceedings of the IEEE International Conference on Services Computing (SCC '06)*, pages 52-59, Chicago, IL, September 2006.

[41] L.A. Zadeh, "Towards a theory of fuzzy information granulation and its centrality in human reasoning and fuzzy logic," in *Fuzzy Sets and Systems*, vol. 90, issue 2, September 1997, pp. 111-127.

[42] R. Schwartz, T. Christiansen (1997) *Learning Perl*, 2nd Edition, O'Reilly and Associates, Sebastopol, CA, 1997.

[43] The Apache Derby Database Project (date accessed: 23 May 2009) http://db.apache.org/derby/

[44] I. H. Witten and E. Frank (2005) *Data Mining: Practical machine learning tools and techniques*, 2nd Edition, Morgan Kaufmann, San Francisco, 2005.