Improving Reliability of Large Scale Multimedia Services

by

Mohammed Fadel Shatnawi

M.Sc., Jordan University of Science and Technology, Irbid, Jordan, 1998
B.Sc., Jordan University of Science and Technology, Irbid, Jordan, 1996

Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of
Doctor of Philosophy

in the
School of Computing Science
Faculty of Applied Sciences

Approval

Name:	**Mohammed Fadel Shatnawi**
Degree:	**Doctor of Philosophy (Computing Science)**
Title:	**Improving Reliability of Large Scale Multimedia Services**
Examining Committee:	**Chair:** Keval Vora Associate Professor
	Mohamed Hefeeda Senior Supervisor Professor
	Jiangchuan Liu Supervisor Professor
	Arrvindh Shriraman Internal Examiner Associate Professor School of Computing Science
	Nabil Sarhan External Examiner Associate Professor Department of Electrical and Computer Engineering Wayne State University
Date Defended:	**November 15, 2018**

Abstract

Online multimedia communication services such as Skype and Google Hangouts, are used by millions of users every day. They have Service Level Agreements (SLAs) covering various aspects like reliability, response times, and up-times. They provide acceptable quality on average, but users occasionally suffer from reduced audio quality, dropped video streams, and failed sessions. The cost of SLA violation is low customer satisfaction, fines, and even loss of business. Service providers monitor the performance of their services, and take corrective measures when failures are encountered. Current techniques for managing failures and anomalies are reactive, do not adapt to dynamic changes, and require massive amounts of data to create, train, and test the predictors. In addition, the accuracy of the these methods is highly compromised by changes in the service environment and working conditions. Furthermore, multimedia services are composed of complex software components typically implemented as web services. Efficient coordination of web services is challenging and expensive, due to their stateless nature and their constant change. We propose a new approach to creating dynamic failure predictors for multimedia services in real-time and keeping their accuracy high during run-time changes. We use synthetic transactions to generate current data about the service. The data is used in its ephemeral state to create, train, test, and maintain accurate failure predictors. Next, we propose a proactive light-weight approach for estimating the capacity of different components of the multimedia system, and using the estimates in allocating resources to multimedia sessions in *real time*. Last, we propose a simple and effective optimization to current web service transaction management protocols.

We have implemented all the proposed methods for failure prediction, capacity estimation, and web services coordination in a large-scale, commercial, multimedia system that processes millions of sessions every day. Our empirical results show significant performance gains across several metrics, including quality of the multimedia sessions, number of failed sessions, accuracy of failure prediction, and false positive rates of the anomaly detectors.

Keywords: Real-time Failure Prediction; Quality of Service; Multimedia Services QoS; Multimedia Capacity Planning; Real-time Transaction Control, Real-time Anomaly Detection.

Dedication

To my mother and father. God blessed me with *ALL* good through you. May God bless you with the highest rewards and honors.

Acknowledgements

I can't express enough thanks, gratitude, and appreciation to my senior supervisor, Professor Mohamed Hefeeda. Your amazing intelligence, dedication, hard and smart work, and genuine care for your students, dear sir, have been an endless inspiration. This research would not be possible without your invaluable guidance and example.

I wish to express my sincere thanks to my examiners and research committee for your help and direction.

To all my honorable teachers throughout the years, I am forever indebted to each and every one of you, for your selfless efforts.

To all my friends and colleagues, from whom I learned humility, respect, appreciation of knowledge, and good work, I thank you most sincerely. Special thanks to the amazing colleagues at the Network Systems Lab.

Last but definitely not least, my deepest and endless gratitude to my beloved wife, amazing kids, and great sisters. I learn from you every day. Your endless love, patience, and support make the journey more rewarding than I could ask for.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we introduce the problem space we are addressing in the multimedia communication services. We provide a brief description of the use cases we focus on, the typical problems facing these use cases, and we summarize the contributions of this research as well as its organization.

1.1 Overview

Online services have Service Level Agreements (SLAs) covering various aspects of the service such as reliability, response times, and up-times. For example, Amazon has a stated up-time of 99.95% SLA, and 3Tera has a 99.999% availability SLA. The cost of not meeting these SLAs is not only low customer satisfaction, but a heavy price tag due to fines and loss of business. It is estimated that the annual downtime cost of IT systems in North America is about $26.5 billion [1].

We focus on multimedia services, as a class of online services, where real-time Quality of Service (QoS) has the most impact on the success of the service. Examples of online multimedia services include Skype, Google Hangouts, and WhatsApp. An online multimedia service is composed of client applications and online cloud infrastructure deployed in multiple data centers around the world, and it uses the Internet as the backbone for communications, as illustrated in Figure 1.1. When a user calls another, a client application initiates the session and invokes an online service endpoint in one of the data centers. This endpoint manages the session by invoking many other sub-services, referred to as components, including identity verification, call management, media adaptation, session routing, experimentation, and advertising. Instances of these components are created and provisioned on different data centers. The perceived quality of the session is directly impacted by the performance of these components. Therefore, if we monitor these various components in real-time to assess their current performance and predict any failures before they happen, we can significantly improve the quality observed by users. This can be achieved

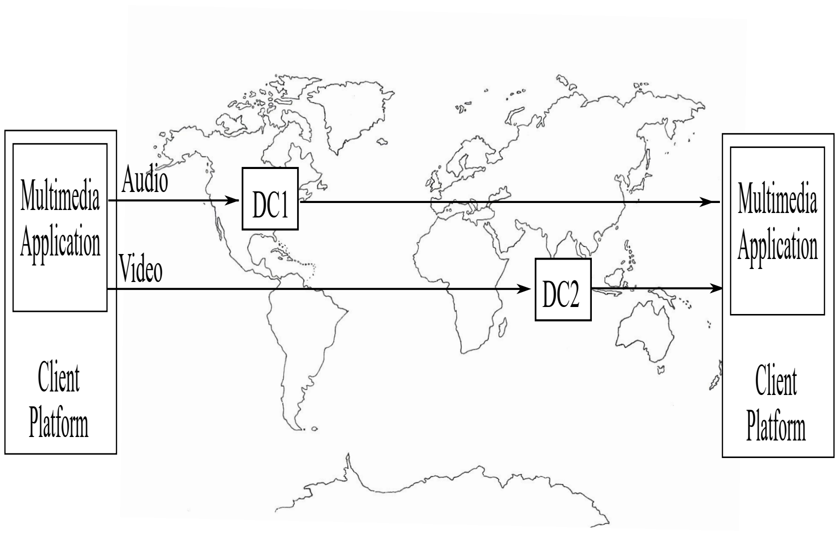

Figure 1.1: High-level architecture of multimedia communication services.

by routing the sessions and multimedia content through the components and data centers that currently have the highest performance and reliability.

The functions of large-scale multimedia communications services, such as user authentication, telemetry, media encoder, dejitter, decoder, storage, and renderer are complex and composed of many software components as illustrated in Figure 1.2. These functions are implemented using various programming languages, run on different data centers, and could be offered by different, internal or external, organizations. One common approach to handle the complexity of large-scale multimedia systems is to design various functions as *web services* that are accessed via standard protocols and interfaces such as RESTful APIs [2]. To serve a request such as create a video conferencing session, the system needs to construct and manage *distributed transactions* involving various web services, while ensuring that resources are not wasted nor over-committed, consistency is always maintained, and concurrent transactions do not interfere with each other. Web services are stateless in nature, which makes handling distributed transactions across multiple web services complex and expensive. This is exacerbated by the dynamic nature and continuous updates of web services [3, 4, 5]. For example, new codecs can be added as new web services, which a multimedia communication service needs to consider without breaking the client code calling the multimedia system.

A key component in the QoS of multimedia services is managing failures in real-time. A failure is an observed deviation of a component from its expected behavior [3]. For example, consider some users having a video conference call using Skype, where they share a presentation and exchange messages. Users expect the audio and video streams and the shared presentation to work properly without failures such as intermittent audio, video glitches, and lost slides. Failure management, a term we use to refer to all aspects of dealing

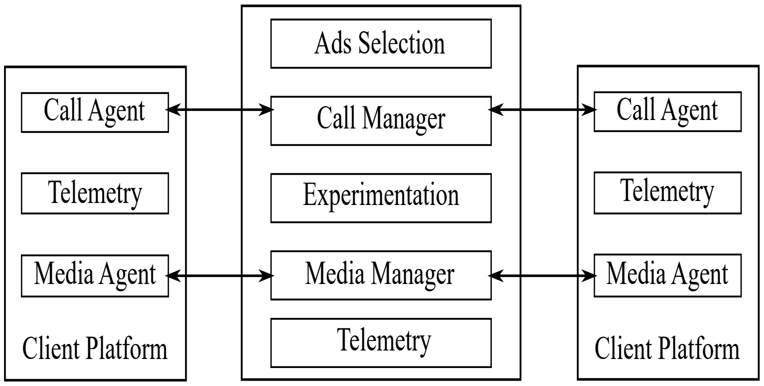

Figure 1.2: Components of a multimedia service.

with failures, plays a key role in the reliability of online multimedia communication services. It includes service monitoring, failure prediction and detection, root-cause analysis, all the way to failure handling and prevention. We focus on end-to-end (e2e) real-time service health monitoring, failure prediction, and making real-time decisions about component selection and routing paths to enhance the reliability and quality of interactive multimedia communication services.

Current approaches for failure management in multimedia communication services, and online services in general, are mostly reactive [6, 7]. For example, Hystrix of Netflix monitors the system for failures in real-time. It has an excellent ability to capture failures and prevent them from propagating and impacting the rest of the system. However, it captures the failures after they happen and impact the system. The creation of a failure predictor is complex and time consuming, and is not suitable for real-time management [8, 9]. The same approaches are used for internal service capacity management. Capacity planning and management are actually part of the e2e fault to failure management. Therefore, the generation of failure predictors and capacity estimators take place before the run-time lifecycle of the online service. The resulting predictors and capacity estimators are built for certain service configurations and working conditions like the number and performance of audio de-jitters and video transcoders. If these configurations change, the predictors and capacity estimators may no longer be accurate. For example, a predictor and capacity estimator could be designed for a video encoder that is able to encode N videos per minute and it fails beyond that. If the administrator adds more resources to the encoder so that it can handle $2 \times N$ videos per minute, the predictor would likely continue to predict failure if the number of videos per minute approaches N, not $2 \times N$, and the capacity management system would not be able to route video calls effectively to these new services. In this work, we call these predictors and capacity estimators **static**, because they do not adapt to changes in the service functionality or the provisioned resources.

In addition, current approaches for online transactional completeness, correctness, and efficiencies such as the protocols in [10, 11, 12], and their implementations in the OASIS projects [13] for managing distributed transactions in web services are not efficient, can lead to substantial waste of resources, and result in reduced multimedia session capacity and quality. The inefficiency is mostly due to a limitation in current protocols that prevents the system from selectively adding and removing individual web services in a distributed transaction without incurring high overhead. To accommodate the dynamic and statelessness nature of web services, current protocols may make the system include unnecessary web services in each distributed transaction. Unnecessary web services for a transaction are those that will not contribute to the successful execution of that transaction. Since web services do not implement transaction rollback [13], the system must issue compensating transactions to reclaim the unused resources and maintain consistency. Compensating transactions are difficult to implement and take long time in real scenarios [13], which result in higher client latency, reduced multimedia session quality, and higher failure rates.

1.2 Thesis Contributions

In the following, we summarize the contributions of this thesis [2, 3, 14, 15]. The evaluation of the thesis contributions is done on actual deployments of online services. The service environment and evaluation setup are described in details in each of Chapters 3 through 6.

1.2.1 Real-time Online Service Failure Predictions

Creating online failure predictors, including data generation, data mining, predictive analysis, training, testing, and deployment are complex, costly, and lengthy operations. Because of that, these operations are done in the offline cycle of online services. To do these operations during the real-time portion of the service lifecycle requires these operations to be focused, and done in a short amount of time. The accuracy of the resulting predictive models is highly compromised by changes that affect the environment and working conditions of the predictor; hence, it is not sufficient to create failure predictors in real-time, but an effort is needed to keep the predictor up to date with changes in the service in real-time.

Online service failure prediction faces the problems of collecting relevant data about the current deployment of the service, building real-time correlations between the inputs, system states, and outputs, and keeping the online failure predictor up to date with the changes in the service. The state of the art approaches to addressing these problems rely on gathering failure data from the logs of the service and building predictive models using that data [7].

We present a new approach to creating dynamic failure predictors for online services in real-time and keeping their accuracy high during the service's run-time changes. We use synthetic transactions during the run-time lifecycle to generate current data about the service. This data is used in its ephemeral state to build, train, test, and maintain an up-to-date failure predictor. We implemented the proposed approach in one of the large scale ad services run by Microsoft that processes billions of requests each month in six data centers distributed in three continents. We show that the proposed predictor is able to maintain failure prediction accuracy as high as 86% during online service changes, whereas the accuracy of the state-of-the-art predictors may drop to less than 10%.

This work has been published in the IEEE INFOCOM conference (INFOCOM'15) [3].

1.2.2 Capacity Estimation for Multimedia Services

Building on the real-time failure prediction work for online multimedia communication services, we address the problem of improving the QoS in multimedia services in real-time. Similar to current failure prediction approach in current online service, capacity planning for online multimedia services is done offline due to the high cost of its creation and maintenance.

Online multimedia communication services go through many changes during their run-time lifecycle that are hard to predict in advance. These changes cause the existing schemes of the service capacity management, load balancing, and traffic routing to go out of date. This results in many failures observed by customers that can be avoided if the service capacity plans are maintained up to date with the latest changes in the service. The state of the art approaches to multimedia service capacity management are reactive in nature, and are built using log data that represent previous deployments of the services. In many cases those logs are no longer relevant to the current service, which make the capacity models created based on them irrelevant and ineffective [16].

To address the problem of enhancing the QoS of multimedia services in real-time, we present a novel *proactive* approach for estimating the capacity of different components of the system and for using this capacity estimation in allocating resources to multimedia sessions in *real time*. The proposed approach is called Proactive QoS Manager. We implement the proposed approach in one of the largest online multimedia communication services in the world and evaluate its performance on more than 100 million audio, video, and conferencing sessions. Our empirical results show that substantial quality improvements can be achieved using our proactive approach, without changing the production code of the service or imposing significant overheads. For example, in our experiments, the Proactive QoS Manager reduced the number of failed sessions by up to 25% and improved the quality (in terms of the Mean Opinion Score (MOS)) of the succeeded sessions by up to 12%. These improvements are achieved for the well-engineered and highly-provisioned online service examined in this work; we expect higher gains for other similar services.

This work has been published in the ACM Multimedia conference (MM'15) [14].

1.2.3 Dynamic Anomaly Detection in Multimedia Services

To complete the multimedia service reliability work, we address the problems facing multimedia service input anomaly detection in real-time. Similar to multimedia service failure prediction, capacity estimation, and transaction web service coordination, multimedia service input anomaly detection is usually done offline. In addition to the reactive nature of this approach, the key problem with current anomaly detection and handling techniques is that they have fixed reaction to anomalies and cannot adapt to service changes in real-time. In current techniques, historic data from prior runs of the service are used to identify anomalies in the service inputs like number of concurrent users, and system states such as CPU utilization. These techniques do not evaluate the current impact of anomalies on the service. They may raise alerts and take corrective measures on these anomalies even if the anomalies do not cause SLA violations. Alerts and their corrective measures are expensive from a system and engineering support perspectives, and should be raised only if needed.

Multimedia communication services utilize advanced protocols that can detect and identify anomalies in their inputs and system states. The problem with existing protocols is that they do not identify nor study the impact of the anomalies on the system before alerting and taking corrective measures when an anomaly is encountered. This results in many false positives in the system. If the service provision changes, which happens often in modern online services, anomaly detectors do not get updated in real-time. This may result in higher rates of false positives and/or false negatives, and thus high waste in the service resources. The state of the art approaches to managing anomalies focus on producing more accurate anomaly detection protocols that favor recent observations and the their observed periodicity, but they do not study the impact of the anomalies on the current deployment of the service [17, 18].

We propose a dynamic approach for handling service input and system state anomalies in multimedia services in real-time, by evaluating the impact of anomalies, independently and associatively, on the service outputs. Our proposed approach alerts and takes corrective measures if the detected anomalies result in SLA violations. We implement the proposed approach in a Microsoft Skype Data Services, and we show that the proposed approach is able to reduce the number of false positives in anomaly alerts by about 71%, reduce false negatives by about 69%, enhance the accuracy of anomaly detection by about 31%, and enhance the media sharing quality by about 14%. The recall in the data generated by the synthetic transactions is 100%.

This work has been published in the ACM Multimedia Systems conference (MMSys'18) [15].

1.2.4 Efficient Web Service Coordination for Multimedia Services

Efficient coordination of web services is challenging and expensive, due to the stateless nature of web services, and because web services change over time. The existing protocols implementing web service transactions are inefficient. They waste resources due to their inability to selectively add/remove individual web services in transactions without incurring high overhead that affects the quality of multimedia sessions.

Online multimedia services tend to defensively include many web services in the transaction, even if some of those web services are not need in the transaction. This results in a high number of unnecessary compensating transactions when failures are encountered. The result is wasted resources and higher number of session failures. The state of the art efforts attempt to mask the problem by optimizing existing protocols to make them faster and less error prone [13].

To address the problem of efficient coordination of multimedia web services in real-time, we propose a simple and effective optimization to current web service transaction management protocols that allows individual web services to *selectively* participate in distributed transactions they contribute to. We implemented the proposed approach in one of the largest multimedia communication services in the world, and found that it enhances the throughput of transactions by 36%, reduces failure rate by 35%, improves multimedia quality (Mean Opinion Score (MOS)) of succeeded transactions by 9%, and reduces the overall time required by all transactions by 35%.

This work has been published in the ACM Network and Operating Systems Support for Digital Audio and Video Workshop (NOSSDAV'16) [2].

1.3 Thesis Organization

The chapters of this thesis are organized as follows. In Chapter 2, we present a brief background on online service reliability, and the classes of problems we address in this thesis. In Chapter 3, we propose a dynamic approach for creating, training, and maintaining an online service failure predictive model in real-time. In Chapter 4, we utilize the principles we introduce in Chapter 3 to estimate the maximum capacity of online multimedia services in real-time, and we build a capacity map that advises the service on the optimum load distribution per service component to avoid online service failures. In Chapter 5, we study the impact of anomalous multimedia service inputs and system states on the output of the service. We build a map of impactful anomalous content that results in SLA violations. In Chapter 6, we introduce the concept of dynamic online web service coordination to allow the multimedia service components to selectively join online distributed transactions that implement e2e scenarios like video calls. We conclude the thesis and discuss potential future work in Chapter 7.

1.4 Summary

In this chapter, we introduced the quality of service (QoS) challenges facing online multimedia communication services. We described a typical use case of multimedia services and the potential points of failures in the e2e system. We summarized the problems facing each aspect of online multimedia communication services and the state of the art approaches to addressing them. We pointed out the specific challenges we focus on in this thesis, and we summarized our contributions.

Chapter 2

Background

In this chapter, we present the basics of online services and their reliability, as well as introduce the key state-of-the-art techniques, in research and industry, for addressing online service reliability issues. This includes a brief introduction to the types of online and cloud services, the classes of service reliability problems, and the key approaches to managing failures in realtime.

2.1 Online Services

Online services refer to the class of software functionality that is provided and hosted on the cloud. Online services cover almost all aspects of used software in today's world. Examples include multimedia communication services like Skype and WhatsApp, social services like Facebook, YouTube, and Twitter, financial services like online banking and credit card services, travel services like Expedia and Travelocity, academia services like online universities, and Customer Relationship Management (CRM) services like SalesForce and Mirosoft Dynamics. These services can be accessed directly using generic web browsers, or through client software apps running as mobile, desktop, and/or web applications. These access modes as well as the type of functionality provided by the online service, define to the type of cloud computing used. **Infrastructure As A Service (IaaS)** offers services like Virtual Machines as an abstraction of physical resources (e.g., computing devices and storage). Sample providers include Oracle VM [19] and Microsoft Hyper-V [20]. These providers offer computing machines like Windows 10 Server and Linux Ubuntu Server. The goal of IaaS is to provide physical resources that can host operating systems and applications without having users actually own nor manage any of the physical components.

Platform as a Service (PaaS) is another class of cloud services. The key offer in PaaS is an abstraction of computing and development platforms; for example, the combination of IaaS and operating system, programming language, and database server. Sample providers include Amazon Web Services (AWS) [21] and Microsoft Azure [22]. They offer platforms like Windows 10 server, Visual Studio 2015 Enterprise, and SQL Server 2016. The goal of

PaaS is to enable developers to build software services without worrying about owning nor maintaining the platform hosting the service.

The last cloud service we cover is Software as a Service (SaaS). The key offers in SaaS include specialized software functionality like multimedia communications. The service is hosted on the cloud, and provided on-demand. SaaS enables customers (enterprise and consumer) to use/purchase services conveniently from any place where there is Internet connectivity. Users call the SaaS functionality, and request the specialized service, like making a call to another user. To provide a bit of motivation for the importance of the area of SaaS, it is estimated that the sales of SaaS in north America will be in excess of $300 billion by end of 2018 and will continue to grow year after year, as reported by Gartner [23]. The estimated number of customers of social media alone, as a class of SaaS, is more than 2 billion users. SaaS provides an abstraction of specialized business processes like banking system, retail shop, travel agency, and multimedia communication. Sample SaaS providers include Skype, YouTube, Amazon, and Expedia.

In this thesis, we focus on SaaS, and take Multimedia Services like Skype and WhatsApp as representatives of this class of online services. A typical interactive multimedia communication service includes client applications and an online cloud service deployed in multiple data centers around the world, and it uses the Internet as the backbone for communications. A high-level diagram of an interactive multimedia communication service is depicted in Figure 1.1. When a user calls another user, the session is routed to the closest data center. The session may be composed of audio, image, and video streams. Each data center implements an instance of the multimedia communication service that is able to handle all streams in a session. Each data center implements a service routing and selection component. If the data center is able to handle the session, it routes all aspects of the session to its internal components. If the data center is able to handle part of the session, e.g., the audio but not the video stream, it routes the video stream to another data center. Thus, a multimedia communication session may have all its streams going through the same path from the source to the destination, or may have some of the streams of the session go through different paths.

Before we end this section, we describe the most commonly used mechanisms in online service communications, and call out the mechanism we will use throughout our research. To call SaaS services, clients use one of the following Application Programming Interfaces (APIs): (1) Remote Procedure Calls (RPCs), (2) Messages, or (3) Resource APIs.

In **RPC APIs**, clients, or applications running on end user devices, call the multimedia services by executing a remote procedure over the HTTP protocol; for example a client may call authentication services, pass encrypted user ids and passwords, and wait for a procedure response indicating successful authentication and authorization. In **Message APIs**, known as Document APIs, clients exchange messages with remote systems without coupling to remote procedures, like RPC systems. Voice transcription from wearable devices is an example of this type of communication. The prominent technology in such a transport mechanism is SOAP and XML, as well as WSDL and XSD. Lastly, in **Resource APIs**, clients provide and consume objects managed by the remote service. For example, a communication application makes a video sharing resource API call, to share a video with another using similar client application. The prominent technology used in this stack is REST APIs. REST is becoming the de facto technology in the majority of SaaS services. We use REST APIs to call all services we work with.

2.2 Online Service Reliability Types

Online service reliability covers both functional and non-functional areas of the service. Functional areas include the actual functionality of the service like buying a product online and making a video call between users. The non-functional areas include all other aspects of the service like availability, scale, and performance.

Functional reliability covers two main concerns: (1) Correctness and (2) Completeness. For example, functional reliability ensures the multimedia video call between two clients takes place correctly between the two intended clients. Service correctness and completeness are generally addressed through software quality assurance and testing. This includes unit tests, build tests, stress tests, functionality tests, multithreading safety, concurrency and access control, authentication and authorization, and debugging tools.

Non-Functional reliability covers three areas: (1) Availability (e.g., Uptime, Downtime), (2) Scalability (e.g., Number of Users, Requests per Second), and (3) Performance (e.g., Response Time, Video Quality). SaaS online service providers define Service Level Agreements (SLAs) with their customers covering all aspects of non-functional reliability. A typical SLA includes clauses like Service Availability with guaranteed uptime of 99.99%, 100 concurrent users, video quality of MOS 4+, lag time of up to 1 second, and response time of up to 250ms. As an example, here is the service level agreement of Google's G Suite [24].

In this thesis, we focus on the non-functional reliability aspects of the service. The importance of this area cannot be over stated. The implications of poor service reliability are high fines, customer dissatisfaction, and potentially loss of business. It is estimated that in North America alone, the cost of SaaS downtime is in order of tens of billions of dollars annually [25].

2.3 Classes of Online Service Problems

Given our focus on the non-functional aspects of the service, we present the four classes of problems in this area, as found in research and industry [6]:

- **Fault**: A fault is a problem with a device that makes it behave unpredictably. A fault can be temporal, transient, intermittent, or permanent. A bad hard disk is an example of a fault.

- **Error**: An error is the direct consequence of a fault. Errors are the deviation from the expected state. Errors can be detected or go undetected. Errors are usually represented as error messages. For examples, attempting to access a faulty hard disk results in a "hard disk not available" error message.

- **Symptom**: A symptom is the out-of-norm behavior of a system parameter. Symptoms are caused by errors. Symptom examples include disk save operation taking too long.

- **Failure**: A failure is the deviation from the expected/correct service behavior and is observed by the user or dependent systems. SLAs are provided to define what users should see as a failure. SLAs are defined for the services output, not the internal implementation or internal state of the service that is not observed or discoverable by an end user. Examples of failures include video lag time more than one second or video quality below MOS 4, if the SLA defined for lag time is sub-second and video quality is MOS 4 and 5. In this work, failure is used to refer to these types of SLA violations. In other words, the service itself could function correctly, but it violated the SLA with its customers for any aspect of its non-functional reliability.

2.4 Failure Management

In a nutshell, the goal of every failure management system is to operate the service with no failures. This is done by preventing, avoiding, and/or hiding failures from being observed by the end users. The end user can be either a customer or a dependent system that consumes the services.

Failure management includes service monitoring, anomaly detection, failure prediction, and failure prevention through avoiding or hiding.

2.4.1 Service Monitoring

Service monitoring refers to the techniques used to quietly read the system state and activities. Service monitors compare the service input, system state, and service output values they read with benchmark/reference values to determine SLA violations. The reference values are found from previous runs of the service, or defined by the rules and SLAs of the service.

Service monitoring provides situational awareness about the service, i.e., what is happening in the service and its components now. The service and its components are called through their REST APIs to measure the system states like CPU and memory utilization, and service state like number of open sockets. Service monitoring can be used for all aspects of service reliability, but it is most commonly used to monitor the symptoms which are the side effects of uncharacteristic functionality of the service.

Many research threads stress the criticality of real-time monitoring of service components. Pietrantuono et al. [3] argue that the ability to monitor a system at runtime and to be able to give estimations about its dependability trend is key to implementing strategies aiming at predicting, and thus proactively preventing system failures. Cotroneo et al. [14] suggest the use of fault injection to make sure that no faults go undetected is a sound way of monitoring. They consider this as a proactive approach and the goal is to improve production log effectiveness. Salfner et al. [26] stress the need for monitoring as a key mechanism to know what is happening in the service in real-time.

It is hard to achieve real-time monitoring. Many systems in research and industry are classified as real-time even though they use logging and offline reporting. By using logs, the process of finding failures and then taking actions requires at least hours if not days [3]. Logs are complex and hard to mine, and it takes hours for the analysis and results of the data to be available for use. Logs are voluminous and are not helpful in many cases because of their large scale [8]. Preprocessing logs to get them to a stage where they are usable in prediction models is tedious and expensive [8, 27].

A key problem with approaches using logging for monitoring is that they are reactive; i.e., failures need to happen first before action is taken. Realizing that failure is the norm in massive online service infrastructures, Hystrix of Netflix [7] aims at isolating points of access to remote systems, services and 3rd party libraries, to stop the cascading of failures. Salfner et al. [26] report that monitoring symptoms and failures is a simple operation, and to be effective it cannot be built on heavy analysis and reporting work in logs, as that will take them to the offline realm, hence the need for failures to occur to accurately report them.

2.4.2 Anomaly Detection

Anomaly detection aims at identifying the service inputs that do not conform with the expected pattern, and finding the values of the service inputs, component working conditions, and system states that deviate from the expected values. The expected values can be found from prvious runs of the service or by the rules of the service [17]. For example, an ad in a video call is expected to last a few seconds before it is registered as a billable impression. If ad impressions are swapping at rates of more than one ad per second then that's an anomaly that indicates ad fraud, and warrants alerting and taking corrective measures such as blocking the ad source.

Current approaches for anomaly detection range from mathematical and data-driven machine learning approaches to system and implementation-based methodologies. Chandola et al. [17] present a survey of the available anomaly detection techniques and their applications. Anomaly detection based on machine learning techniques use either: (1) historic data about the system at hand, or (2) rule-based approaches. The output of the state of the art anomaly detection techniques used in online services is in the form of a set of static boundary conditions on the service inputs and its system states [14]. Outliers in such models are considered anomalies even if they do not result in any scenario failures. The key issue with almost all machine learning approaches is their dependence on large amounts of data to create, train, and test new models [14].

The use of system logs for anomaly detection entails high cost. The cost is associated with preprocessing the data. The data preparation time is generally too high to make these approaches suitable for real-time changes and updates [3]. The majority of anomaly detection using machine learning approaches rely on service production logs to find the ranges of service inputs and system working conditions that can be studied to identify the outlier boundaries [17]. Data in logs may not be sufficient for mining, analysis, and anomaly detection models [26]. The resulting anomaly detection models created based on logs from prior runs may not accurately represent the current system at hand [14].

The reactive nature of using logs in anomaly detection is a problem. Even with online system-monitoring-based approaches, like that used by Hystrix of Netflix, the monitoring is still reactive, as the anomalies and failures need to happen and customers endure them before they are controlled [7]. Leners et al. [28] use failure informers to improve availability of distributed online systems; but these are reactive in a sense, as they are built using system messages found in production logs from prior runs of the service, not from the current service in real-time. System-based approaches for real-time service failure prediction [3], and service capacity estimation [14] are used to address the problem of getting real-time data that represent the current service, and they use that data to predict service failures and SLA violations.

2.4.3 Failure Prediction

Failure prediction aims at anticipating or forecasting the occurrence of event values that constitute an SLA violation; i.e., an output that deviates from the expected service output. By predicting future failures before they happen, services can take actions to prevent them. Failure prediction focuses on the service output events that constitutes an SLA violation, like Time to Add Product to Cart, Ad Rendering Time, and Travel Ticket Booking Time, and attempt to prevent them.

The majority of failure prediction techniques found in the state of the art literature and industry applications are based on production logs, just like those techniques used for anomaly detection and service monitoring. Data generation is mainly done through execution of real transactions, and data collection is through writing the transaction and its results to logs [8]. Li et al. [4] propose WebProphet and the use of parental dependency graph to encapsulate web object dependencies to implement web page load predictions. The data generated and collected for WebProphet is done through service logs. Viswanathan et al. [29] develop semantic framework for data analysis to enhance the performance of networked systems, and logging is the mechanism of collecting data about the system. We note that all efforts related to our work of failure prediction depend on some form of logging for later analysis, and act on failures after they happen. Considerable effort is put to enhance the performance of mining the logs [3].

Data mining and machine learning techniques used in creating failure predictors require data to be in some structure [8]. Many efforts have focused on enhancements of production logs by adding structure through the use of meaningful logging. Zheng et al. [30] suggest event categorization and filtering of logs to overcome their lack of structure and lack of usability for data mining. Xu et al. [8] attempt to identify problems with production logs of distributed systems, and suggest methodologies to enhance the performance of mining the logs by automatic matching of log statements. Cohen et al. [31] describe how failure prediction models are built to identify and study the root-causes of failures. They propose techniques to categorize the faulty execution results found in the logs, before building failure prediction models based on them. They propose the use of indexing and clustering of system histories to correlate with failures.

To perform root-cause analysis, failure predictors aim at analyzing the execution path structures that lead to failures. This is done by using instrumentation data from online servers to correlate bad performance and resource usage. Sambasivan et al. [32] implement path tree comparisons, comparing and cross-correlating data from different sources in the service, as means of predicting the paths that lead to failures. This effort is expensive and hard from service logs.

2.5 Summary

In this chapter, we presented a brief background on online services, taking multimedia communication services as the emblematic case study. We described the basics of online service designs and models, the types of reliability challenges facing online services, the classes of online service problems. We described online service failure management approaches and summarized the key efforts found in each area. In the following chapters, we will focus on each one of these failure management problems, and present our work to solve it.

The scope of this research is the real-time analysis and management of online service failures. As such, it doesn't cater for deeper or longer analysis of failure management. The latter is part of the analysis and management done using the holistic data set from the service logs. The real-time components of the failure management aim at identifying the current and trending conditions that lead to SLA violations in specific scenarios of interest, like video quality.

Chapter 3

Real-time Failure Prediction in Online Services

As described in Chapter 2, failure management addresses the areas of service monitoring, failure prediction, and anomaly detection. In this chapter, we focus on failure prediction in the run-time lifecycle of the service. We present the key approaches for failure prediction in online services, discuss their shortcomings, and propose a new approach to addressing these shortcomings.

3.1 Introduction

There are several approaches to create an online service failure predictor. These include statistical methods like Bayesian, decision rules methods like decision-trees, artificial intelligence methods like neural-networks, and cluster analysis methods like clustering algorithms [33, 34]. The predictor generation entails: (1) generation of data about the system; including its inputs, working environment, and outputs, (2) collection of this data into containers like log files to be used later, (3) pre-processing and analysis of the data to exclude extraneous data, and organize the remaining impactful data into usable data models like dimensional models [35], (4) designing of prediction algorithm(s) and system(s), (5) training the predictor, (6) testing it, and (7) deploying it in production to be used in failure prediction in the online services. These steps are lengthy, complex, and time consuming [8, 9]. Thus, they usually take place before the run-time lifecycle of the online service. The resulting predictor is built for certain system configurations and working conditions like the amount of available resources and their performance characteristics, e.g., the number of routers and their throughput, and the number of database systems and their sizes. If these configurations change, the predictor may no longer be accurate. For example, the predictor could be designed for a database system that meets its throughput SLA when it encounters up to N concurrent requests per second, and it fails beyond that. If the system administrator adds a performance enhancing database cluster to the database system where now it can handle up to $3 \times N$ requests per second and still meets its throughput SLA requirements, the predictor would likely continue to predict failure if the requests per second approaches N not $3 \times N$. In this chapter, we call these predictors **static predictors**, because they do not adapt to changes in the service functionality, the system resources, or other changes.

Furthermore, data mining techniques used to generate failure predictors require high volumes of data to reach high prediction accuracy [8, 27, 34]. Current data mining techniques de-emphasize the system under study, by treating it as a black box, and focus on its inputs, outputs, and working conditions to build the failure predictor models. These characteristics of current data mining techniques make them not suitable for real-time creation and updates. Static predictors work well within environments that do not change often; such as transportation systems like airplanes and navy ships, engineering systems like factories and assembly lines, and software systems like games. On the other hand, modern online services lack such stability over time at many levels including functionality, designs and implementations, and service hardware provisions to accommodate the changing user requirements and loads over time. The ever-changing landscape of online services, coupled with requirements such as continuous up-times, make the use of static failure predictors challenging and less efficient.

We propose a new approach to failure prediction in online-services during their real-time lifecycle that overcomes the problems noted above. Real-time refers to the runtime lifecycle of the service, where the service is in production and is being used by real customers. We use synthetic transactions during the service real-time lifecycle to generate current data about the service. This data is used in its ephemeral state to build, train, test, and maintain an up-to-date failure predictor. We evaluate the effectiveness of the proposed approach on a large-scale enterprise backend ad service. The service handles over 4 billion ad requests a month. We show that during the production phase where the service goes through changes, our approach is able to maintain high prediction accuracy average of 86%, whereas the prediction accuracy of current state-of-the-art predictors may drop to less than 10%. The recall of the data generated by our synthetic transactions is 100%. Recall in this context refers to the percentage of the generated data that is relevant and used. In contrast, the recall in the production logs is less than 2%. In addition, we show that we can update the predictor in real-time in less than 7 minutes; this includes generating data, creating the predictor, training it, and testing it. On the other hand, building a failure predictor using typical data mining techniques for the same service by using production logs requires about 5 weeks of production running and logging, and it takes more than 17 hours of pre-processing, training and testing.

The contributions of this chapter are (1) a novel approach to build real-time failure predictors, (2) a light-weight data mining algorithm for failure predictors in online services, and (3) the actual implementation and deployment of the proposed approach in a real online service environment.

3.2 Related Work

We summarize the current approaches for creating failure predictors based on production logs. Then we discuss the need to have failure prediction in online services, and the key efforts done there.

3.2.1 Handling Production Logs

Quite a few research efforts emphasize the problems encountered in building accurate data mining models based on production logs. First, logs are complex and hard to mine [8]. Second, data in the logs may not be sufficient for data mining [8, 27]. Third, pre-processing the logs to get them to a state where they are usable in prediction models is tedious and expensive [8, 30]. Snyder et al. [27] argue that the insufficiency of log data causes problems for mining them because data in the logs are extraneous, and it is hard to identify the relevant pieces that are needed in the data mining process. Xu et al. [8] describe the voluminous nature of data in production logs, and show that logs are not actually helpful in many cases, because of their large volume. Chen et al. [36] study the application of machine learning to logs of faulty executions to predict the root cause of failures. Their Pinpoint model requests paths in the system to cluster performance behaviors, and identify root causes of failures and anomalous performance. Pre-processing of system logs to prepare them for analysis and mining is studied by Salfner et al. [26].

The research threads above describe the problems with production logs that make them hard to work with, especially in real-time. They aim at alleviating some of the problems and symptoms, but come short of reducing the cost of processing production logs to levels that are suitable for real-time analysis and mining. These limitations show the need to produce a real-time predictor that does not depend on production logs.

3.2.2 Online Service Failure Prediction

Data mining and machine learning techniques used in creating failure predictors require data to be in some structure [33, 34]. Many efforts have focused on enhancements of production logs by adding structure through the use of meaningful logging [37]. Zheng et al. [30] suggest event categorization and filtering of logs to overcome their lack of structure and lack of usability for data mining. Xu et al. [8] attempt to identify problems with production logs of distributed systems, and suggest methodologies to enhance the performance of mining the logs by automatic matching of log statements. Cohen et al. [31] describe how failure prediction models are built to identify and study the root-causes of failures. They propose techniques to categorize the faulty execution results found in the logs, before building failure prediction models based on them. They propose the use of indexing and clustering of system histories to correlate with failures. Leners et al. [28] propose an algorithm to improve availability in distributed systems by using failure informers. The failure informer is a reporting service that is built based on mining and analyzing the system messages in the logs.

In addition to root-cause analysis, failure predictors aim at analyzing the execution path structures that lead to failures. This is done by using instrumentation data from online servers to correlate bad performance and resource usage. Sambasivan et al. [32] implement path tree comparisons as means of predicting the paths that lead to failures. Realizing that failure is the norm in massive online service infrastructures, Hystrix of Netflix [7] aims at isolating points of access to remote systems, services and 3rd party libraries, to stop the cascading of failures. This adds resilience to services in distributed systems. Hystrix uses real-time monitoring, and acts on failures after they are detected. A few efforts attempt to build predictive models based on execution analysis by replaying the debugging information in logs [37, 16, 26]. Li et al. [4] propose WebProphet and the use of parental dependency graph to encapsulate web object dependencies to implement webpage load predictions. Viswanathan et al. [29] develop semantic framework for data analysis to enhance the performance of networked systems, and logging is the mechanism of collecting data about the system.

We note that all efforts related to our work on failure prediction depend on some form of logging for later analysis, and act on failures after they happen. Considerable effort is put to enhance the performance of mining logs. Our approach has a key difference from existing approaches to enhancing online service reliability. We use data in real-time, because the cost of working with logs is too high for real-time processing.

3.3 Proposed Approach

Predictors are designed to anticipate the outcome of an event [33, 34]. In online services, events represent a variety of measurable aspects and characteristics of the service, such as the response time of the service, the availability of the service, and the number of packets routed correctly in a given time. The dynamic data prediction approach we propose is usable with any of these events. For the rest of the work, we use the term **failure prediction** to indicate predicting when the outcome of an event does not meet its SLA. For example, if the event of interest is response time, then failure prediction means predicting the cases where the operation under study does not finish within the SLA.

We define two concepts: **Local System** and **Scenario**. Local System refers to the stack of software, hardware, and operating system used to perform a specific functionality in the online service. Scenario refers to the collaboration of the set of local systems used to implement an end-to-end (e2e) user scenario. As an example, assume the online service of interest is a retail service, where customers buy sports products. An example of e2e scenario is finalizing an online purchase through a checkout process. Assume the event of interest is response time. The response time SLA for the e2e transaction could be 300ms;

the purchase process meets its SLA if it completes in less than 300ms, and fails otherwise. Assume that the checkout process makes the following calls behind the scene: check cart contents, check product availability, calculate total price, validate credit card, submit the transaction, return to user, and update all related systems such as inventory, credit cards, and logging systems. The collaboration of these local systems to implement the purchase process represents the checkout scenario.

3.3.1 Testing in Production

Before we delve into the details of our approach, we describe Testing in production (TiP). TiP is a set of software testing methodologies used to call the online service APIs just like real users and other software components do. The testing calls are made regularly to all components of the online service and the results are collected and verified to make sure the e2e service is working properly [38, 39, 40]. Enterprise service providers such as Facebook, Google, Microsoft and Yahoo use TiP to perform functional, stress and performance in real-time [40]. The synthetic transactions used in TiP generate loads that are marked with special moniker(s) so that they are distinguished from real transactions, and do not interfere with the service destructively, or result in an incorrect state of the system like product inventory reduction due to test purchases. Synthetic transactions in TiP do utilize the service resources, and this puts an impact on the service. Service designers account for such impact due to the importance of TiP; without it the service would be flying blind [38, 40]. Figure 3.1 shows the relationship of a TiP system, the Failure Prediction System in the figure, to the production system. The TiP system runs in parallel to the production system and has access to the production system components. However, faults and failures in the TiP system do not impact the production system.

We utilize TiP principles and infrastructures as the platform for our suggested approach. No production code is instrumented to generate the data. Data is generated through TiP synthetic transactions. This is an advantage of our approach, because we do not impact the production code, and thus no extra testing is needed. Updates to the failure predictor do not require production code update or redeployment. We use TiP to cover functional, performance, and data failure modes.

3.3.2 Overview of the Proposed Dynamic Failure Predictor

Our proposed dynamic failure predictor is suitable for the dynamic nature of online services' functionality, environments, and elasticity of their loads over time. It can be used for systems with static environments as well, but it is superior in systems with dynamic situations. The proposed approach can be summarized in the following steps:

- Step 1: Using synthetic transactions, local synthetic transactions (LSTs) and scenario synthetic transactions (SSTs), to execute the local systems of the online service in a

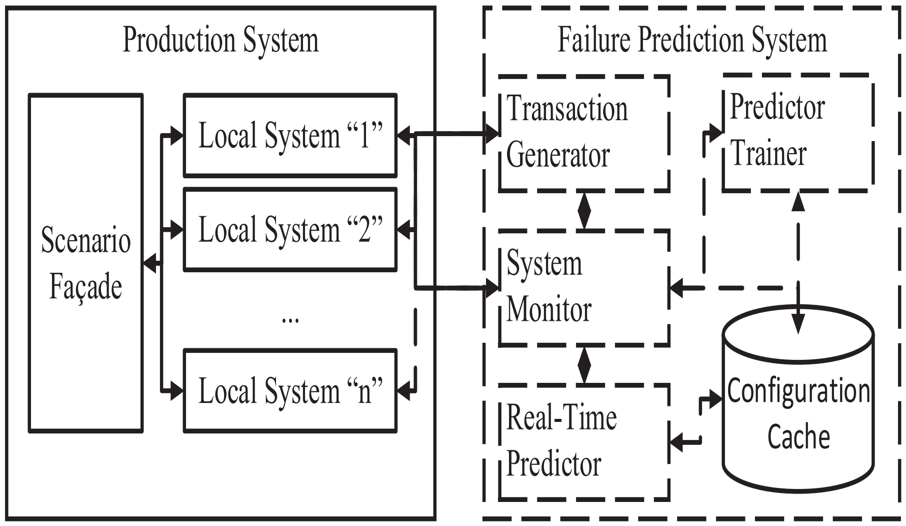

Figure 3.1: Interaction between proposed failure predictor and production system.

way that mimics user behavior, and constitute a complete e2e scenario such as a user buying a product online.

- Step 2: Collecting the synthetic transactions inputs, local systems outputs, e2e scenario outputs, and events of interest. This data is collected into in-memory constructs with a predefined dimensional model [35].

- Step 3: Using the freshly collected data from running the synthetic transactions to correlate the local systems inputs, local systems outputs, the output of the e2e scenario, and the event of interest.

- Step 4: Building a real-time predictor from the collected data and the correlations found in the previous steps, and training and testing it in real-time.

The failure predictor is used in production as long as it has high prediction accuracy. We can measure its prediction accuracy since we know the output of the e2e scenario and the output of the event of interest from running the synthetic transactions as well as real transactions in real-time. If the accuracy of the predictor drops below a threshold for a given period of time, we rebuild a new predictor, train it, and test it.

Figure 3.1 depicts the relationship between a production system and the failure prediction system used to generate and maintain the predictor. The production system is comprised of multiple local systems that constitute one e2e scenario. The failure prediction system is an independent product that runs in the same environment where the online service runs. It has access to the same resources as the production service, but is not part of the online service code. Updates and re-deployments of the predictor code can be done anytime without interfering with the service production system.

3.3.3 Generating Real-time Synthetic Transactions

The transaction generator, in the failure prediction system in Figure 3.1, mimics the user scenario by making all the local system calls that comprise the scenario. The system monitor collects the responses of each LST, the event of interest, and other information about each local system such as the number of running tasks, and the used resources in the system.

The algorithm we propose to generate synthetic transactions is as follows. The transaction generator makes LST calls using test loads, which are similar to real loads. The real user behavior (loads and call distributions) are found from the logs of previous deployments of the service, or from field/market studies about the service. For example, it would be known based on what is found in previous production logs, or estimated from market research, that the average customer of a retail online store buys 5-10 items a time, and that the service gets around 500 concurrent users at peak times. So if local system 1 is an addProductToCart() function, the average expected products to add are 7, and the average expected concurrent function calls are 100. Then the LSTs made by the predictor transaction generator start with these loads, and progressively add more loads until the failure causing loads are found. The test loads are executed on the current system, and so generate current information that represents the current state of the system.

The LST calls are made at equidistant time series; once every N seconds. The value of N is configurable, depending on the service, and it varies during execution time, increases or decreases, depending on the state of the service. If the service is churning and failures are happening more, then N is reduced to get a better pulse of the system. The event is captured after each of these tests, and its value is compared to the result of the prediction. If the accuracy of the predictor starts to drop, the tests are done at a higher rate to generate data to build a new predictor. The set of tests performed every N seconds need not be exactly like those of customer behavior. However, if the service at hand requires an exact replica of the user behavior, then parts of previous production logs representing that behavior can be replayed as suggested by [37, 16, 26].

3.3.4 Collecting the System Data

The system monitor collects the data generated by running the LSTs around the failure points found by the transaction generator. The data is passed to the predictor configuration cache, as shown in Figure 3.1. The data includes the current load and state of the system like the number of tasks, and the used resources like CPU and memory utilization. The data is collected and hosted in memory in an array with a dimensional model schema [35]. Dimensional modeling refers to a set of concepts and techniques used in data analysis which provide insights into the cause-effect relationships between entities. Data is organized into two sets, a set of measured or monitored data, called **facts**, and a set of parameters, called **dimensions** that define, impact, and control the facts.

Dimensions					Fact
LST	Tasks	CPU	Memory	LST time	SST time
1	53	19%	35%	23	191
2	53	17%	35%	17	191
3	53	18%	35%	31	191
4	53	11%	35%	19	191

Table 3.1: Sample of Local System Response Time SLA.

For this work, the fact is the event of interest that is to be predicted, such as response time. The dimensions are the other sets of data that impact the event. The dimensions are generated using the actual LSTs, and their values are found from the responses of each local system, such as:

- The **LST** that is called: this allows us to know which test is run.
- The **time** that LST is called: the time stamp is what allows us to relate and study the cause and effect in the system calls, i.e., at this time there are this many function calls and this many tasks, which caused this response time.
- The **number of tasks**: processes or jobs in the system.
- The **resource (CPU, compute servers, and memory) utilization**: although this may be seen as a measure, it plays the role of a dimension that impacts the event.

Table 1 presents a sample of such a dimensional model. Note that we use surrogate keys [35] to represent non-measured and non-numeric values. A surrogate key is a unique identifier of an entity; it can be an integer and it is not derived. This makes a table with smaller footprint in memory, only integers are used, and enhances the performance of operations done on it.

3.3.5 Design of the Real-time Predictor

The predictor trainer, in Figure 3.1, is the module that creates, trains, and tests the proposed predictor. This happens on the test platform not the production platform. A predictor, fundamentally, is a classification system [33, 34]. It defines and monitors boundaries of working conditions that result in an event. It predicts the outcome of an event based on the system working conditions and loads. Our proposed light-weight predictor defines the independent variables of the local systems that impact the event of interest; the independent variables are the ones that can be controlled. The independent variables we propose to use are the number of active tasks, jobs and processes, in the system. We identify the dependent variables that control the output of the event to be the resource utilization; the compute and memory resources. They are dependent on the independent variables, but their values impact the event of interest.

Our classifier follows a logical expression model that defines the safe working boundary conditions for each of these variables by their upper values. For example, the event of interest would succeed if:

- Condition1:
  - IndependentVariable11<L11
  - ...
- Condition2:
  - DependentVariable21<M21
  - ...
- ConditionN:
  - IndependentVariableN1<PN1
  - ...

As an example, Condition1 may have two independent variables and their values that cause failure as follows: concurrent-processes <15 and Number-of-routed-packets <100. At the time of running the LSTs, the values of concurrent processes, active packets, LSTs inputs, and event value are captured from the system. They are then fed to the predictor. If the value of concurrent-processes is 13, and number of active packets is 95, the predictor predicts that the operations over the coming period of time will be successful, but issues a warning to the load-balancing-system to reduce the system loads as it is approaching failure points. The prediction value (success in this case) is compared with the actual event values that are captured from the system, ground truth, over the coming period of time. This allows measuring the accuracy of the predictor as well as false positives and negatives. We explain how to find the variable values that result in event failures in the next section.

3.3.6 Training and Testing the Predictor

Our approach uses synthetic transactions to execute the system by controlling the load to produce failures. It captures the transactions inputs, the results from the transactions, the event value, and the used loads. We can search for the classifier values that result in failures in the captured data. This is an advantage of our approach, as the findings are based on the ground truth.

The Real-Time Dynamic Failure Predictor (RTDFP) algorithm we devise, to find the classifier values that result in failures, is as follows:

1. Make a new set of LST calls with increasing intensity until the LSTs start to result in violating the SLA of the event.

2. Test the system using LST calls with loads around the failure causing loads.

3. Capture the LST inputs, system states, independent and dependent variable values, and event value into an array with a dimensional schema similar to Table 1.

4. Use search algorithm, e.g., A* search, to find the classifier values that correspond to the event failures.

5. Store these values for each local system in an array with a schema similar to that shown in Table 2. We call this table the Local System Predictor Configuration Table (LSPCT).

6. Repeat these steps, 1 through 5, and capture T instances of Table 2 until the classifier values reach a steady state; a good way to determine that is by using standard deviation for the variable values. The system continues to be in change mode until a steady state in these values is reached. The number of table instances, T, depends on the system characteristics. Some systems reach a steady state faster than others, and may require a low T value like 5 tables. Other systems with higher instability characteristics may require more tables. It is up to the system designers to define T. The T tables extracted from running sets of LST tests constitute a rolling window, i.e., the $T+1$ new LST tests will push out the results of the first LST run so that the tables have the LST test sets 2 to $T+1$.

When a steady state in the classifier values is reached, a new instance of Table 2 is created by averaging the values found in the T tables created during the training steps. This instance is called the predictor configuration table (PCT). These average values, in the PCT, are the values of the classifier variables that result in failures.

To test the predictor, we run a new set of LSTs around the failure points and capture the event value for each. We test the predictor accuracy by comparing its predictions against the event value for each test. If the accuracy of the predictor meets the goal, it is ready for use. If not, we repeat the training steps. When the predictor passes testing, the values in the PCT are stored in the predictor configuration cache, and are used as the classifier values that would predict failures in the system. Procedure 1 illustrates the RTDPF algorithm to train the proposed predictor.

Note that other techniques, such as regression [33, 34], can be used to find the predictor variable values that result in failure. However, we believe that searching the results of the synthetic transactions has better results as the values are found based on the ground truth coming fresh from the system.

Procedure 1 RTDFP Algorithm

```
PREDICTOR TRAINER
 1: function TrainPredictor
 2:     $Counter = 0$
 3:     $T$ = NumberOfTables2                                    ▷ configured value
 4:     $FailurePoints$ = FindEventFailurePoints()
 5:     while No-Steady-State-in-Classifier-Values do
 6:         RunTestsAroundFailurePoints();
 7:         Search for values that cause failure;
 8:         Store values into ($Counter$ % $T$) of Table2;
 9:         Compute steady-state-in-classifer-values;
10:         Increment Counter;
11:     end while
12:     RunTestsAroundFailurePoints();
13:     Test Classifier Values;
14:     if ClassifierValueTest Succeeds then
15:         $PCT$ = Average Table2 $T$ Instances;
16:     end if
17:     if ClassifierValueTest Fails then
18:         TrainPredictor()
19:     end if
20: end function
FIND EVENT FAILURE POINTS
 1: function FindEventFailurePoints
 2:     while no-event-failures do
 3:         Increase LST loads
 4:         Run LST calls
 5:         Capture Event-Failure-Causing-Loads
 6:     end while
 7: end function
TEST EVENT FAILURE POINTS
 1: function RunTestsAroundFailurePoints
 2:     for each $n$% below Event-Failure-Causing-Loads to $n$% above Event-Failure-Causing-Loads
    do
 3:         Run LST calls
 4:         Capture LST and system values into Table1
 5:     end for
 6: end function
```

Local System	Tasks	CPU	Memory
1	153	68%	73%
2	149	71%	68%
3	223	67%	71%
4	196	70%	72%

Table 3.2: Sample Configuration of Local System Predictor.

The accuracy of the predictor is continuously monitored, by the system monitor in Figure 3.1. If the predictor is successful at predicting the event, the predictor is kept. If not, the system is tested until it reaches a steady state, before deciding to create a new predictor. Note that during system changes, the old predictor is kept in use until the new predictor is created. When a new predictor is created, the predictor trainer updates the predictor configuration table, which updates the dynamic real-time predictor without any impact on the service.

3.4 Evaluation

We validate our approach on a large-scale enterprise online ad service, used in Microsoft, that works as an ad request arbitration unit. The high level diagram of the service is depicted in Figure 3.2. The service is composed of the ad request facade, the ad request processor, the targeting system, and the response validator. The service is a large distributed system, has challenging requirements, and operates under strict SLA, making it an ideal environment for the validation of the proposed technique. It is hosted in three continents; North America, Europe, and Asia. It receives ad requests from applications running on mobile devices and PCs. It processes each ad request, determines the source, PC or mobile, and attempts to find targeting opportunities based on app categories, device types, user information if available with user permission and consent to use, and client location. It then makes a call that has the targeting information to an ad serving network. The ad serving network identifies a suitable ad to be served to the application, and sends a response back to the service. The service validates the ad response against rules about the user, the calling application, ad type, and ad size, and returns the ad response to the calling application. The response time SLA for serving an ad is 250ms; this is the event value we watch for.

3.4.1 Implementation and Setup

We implement the LSTs for the ad service local systems and measure the SST response time for the ad request scenario. There are three local systems: ad request processor, targeting system, and response validator. For our study, we use a local ad serving network, which becomes the fourth local system.

The ad service gets an average of 4 billion ad requests a month. It is deployed in 6 data centers, 3 continents with 2 data centers each. The setup we have is based on the model shown in Figure 3.1. The transaction generator makes a call to each of the local systems and controls the various aspects of the ad request, client type, ad type, location, and user information. The transaction generator simulates ad request calls made by multiple clients, by making simultaneous calls with different user agent information. It controls the load in two ways: the number of ad requests made by each simulated client in a given time, and the number of simultaneous ad requests representing multiple client calls. We implement a

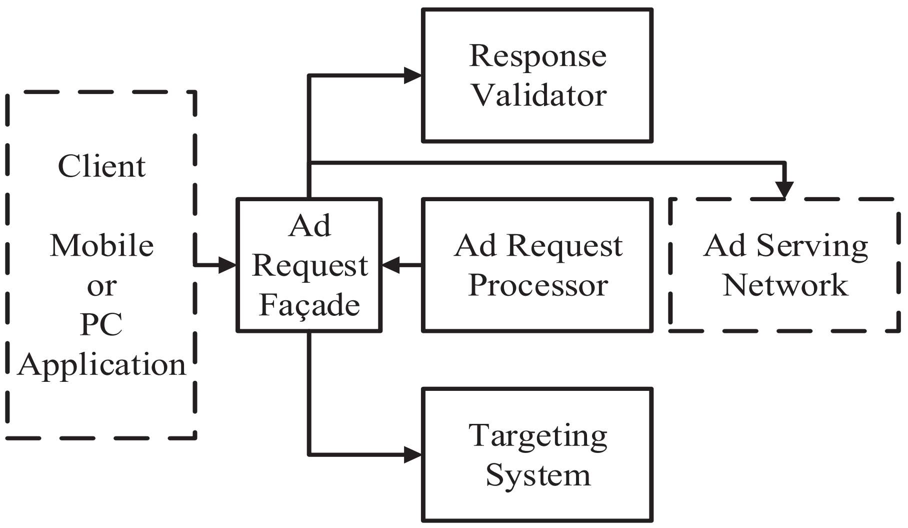

Figure 3.2: The online ad service used in the experiment.

predictor instrumentation that can make up to 10,000 concurrent ad requests. By design, each user cannot request more than one ad every 30 seconds, this is a real requirement to prevent ad fraud. The maximum load that can be generated by the failure prediction system is 10,000 simultaneous ad requests every 30 seconds. This is more than 100x the expected real load of the system; so it is a good stress test. Each ad request results in an average of 0.5KB of data returned from the service, and this is what is captured by the system monitor into a schema similar to that shown in Table 1. The prediction system is implemented and is used for several months.

3.4.2 Data Collection

Responses to the LST calls and each local system state are collected for every LST call made to the service. These values are stored in memory arrays with the schema of Table 1, in the predictor configuration cache. In steady state, when the service is not going through any changes that warrant a predictor update, which is determined if the predictor maintains accuracy above threshold, the synthetic transactions are designed to run 100 concurrent LSTs every minute. Each LST takes an average of 25ms to complete, so running 100 tests every minute takes less than 3 seconds. Note that the LSTs are used for production testing purposes as well as for predictor verification. If the prediction accuracy drops below a threshold, we rebuild the predictor using the RTDFP algorithm. We use a high threshold of 80% accuracy, to ensure the online service maintains its failure SLA. We use the algorithms described in Section 3 to collect new data, create a predictor, train it, and test it. We find that it takes, on average, 3-5 minutes with 7,000 to 10,000 LSTs to generate enough failure information to start the creation, training, and testing of the predictor. Each transaction generates an average of 0.5KB of data, so the total data used in training is 5MB only. In contrast, the production log has an average of 88GB of data a day, and requires 5-6 weeks worth of production running, as shown later in Table 3, to generate enough data that can be used to create, train, and test the failure predictors.

Using synthetic transactions for no longer than 7 minutes we were able to generate data, create, train, and test the proposed dynamic predictor and get to accuracy close to 86%. On the other hand, we find that the average real production failure rate is less than 1%. That is an average of less than 150 failures a minute in the whole data center which has hundreds of servers. There is the fact that production failures do not actually happen uniformly throughout the day. So from a test perspective, it can be hours, or even days, before real failures are encountered.

3.4.3 Performance Metrics

We validate the effectiveness of our approach by measuring its ability to adapt to production system changes, and still maintain high accuracy. We compare the performance characteristics of the proposed predictor with four static predictors based on neural network, clustering, Bayesian, and decision trees algorithms. We do not implement these predictors, we use commercially available software that implement them. We choose these algorithms because of their wide use [36, 31, 8, 32] and because they represent different approaches to data mining and machine learning; they represent Artificial Intelligence, Clustering Analysis, Statistical Methods, and Decision Rules respectively [33, 34, 41].

To assess the accuracy of the proposed dynamic predictor, we use the following metrics:

- **False positives**: a prediction is called a false positive when a transaction is predicted to fail to meet the response time SLA, but the transaction meets the SLA time.
- **False negatives**: when a transaction is predicted to meet the response time SLA, but the transaction fails to meet the SLA time.
- **Accuracy**: is the ability to predict the results of the new transactions correctly; i.e., the predictor's ability to identify and exclude true errors. It is calculated as the ratio of correct predictions, which is 'all predictions' minus 'true errors', divided by all predictions.

3.4.4 Detecting Failures in Dynamic Environments

To test the predictors' ability to adapt to real-time changes, we start with a baseline configuration and configure/program the four commercial static predictors. The proposed dynamic predictor is built during the experiment. The baseline configuration is a cluster of 10 compute servers. Each compute server is a quad-core intel Xeon server with 12 gigabyte RAM. We deploy the predictors into the test client. We run the synthetic transactions, LSTs, and SST. We make the following changes: increase the available compute resources by 5 more

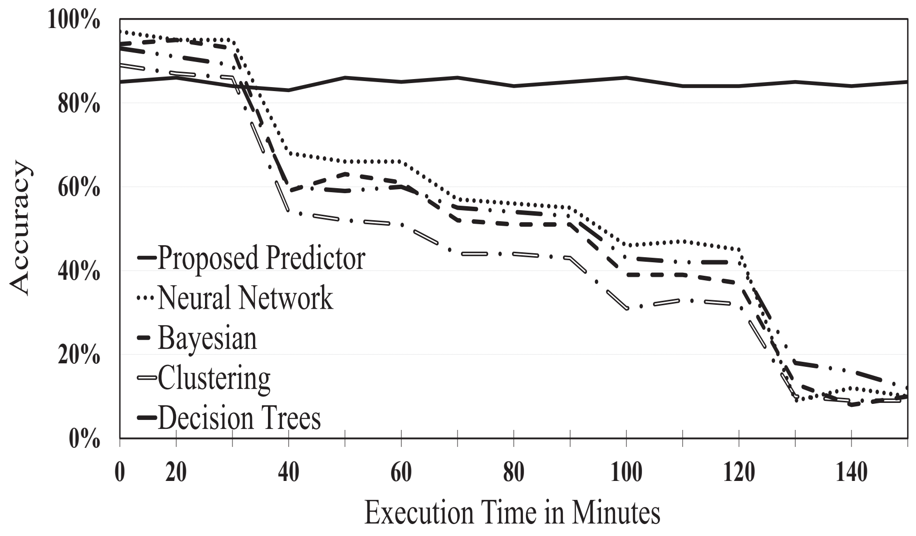

Figure 3.3: Accuracy of proposed vs. current predictors in dynamic environments.

processing servers and wait for thirty minutes, and then increase by 5 more servers for a total of 20 servers, and wait for 30 minutes. We drop the compute resources to 6 servers and wait for thirty minutes, and then drop the servers to 3 and wait for 30 minutes. During that period, we capture, every minute, the results of the LSTs and SST, event value, which is response time for the tests, and predictions made by all predictors. The event value we capture is the ground truth. We then measure the false positive rates, false negative rates, and the accuracy of all predictors to determine how they react and adapt to production system changes.

Figure 3.3 shows the accuracy of all predictors in dynamic environments over 2.5 hour period. Figure 3.4 shows the false positives, and Figure 3.5 shows the false negatives for all predictors. As expected, the current static predictors fail to adapt to the resource changes; they drop in accuracy after the first configuration change, which is adding 5 processing servers, and never regain their accuracy back. It is worth noting that after increasing the compute resources above the baseline, it is the false positives that are responsible for the drop in accuracy. In other words, many transactions are actually successful, but are predicted to fail. While after reducing the compute resources below the baseline, it is the false negatives that are responsible for the drop in accuracy. On the other hand, Figures 3.3 through 3.5 show that the proposed predictor is able to maintain high accuracy. It manages to maintain the same average of false positives and false negatives post the compute resource changes. On average, it takes the proposed predictor an average of 7 minutes to adapt to the changes and reflect current state of the system.

The transient period, which is the period from the time we add or remove resources until steady state in the proposed predictor is reached, is on average 7 minutes. That's how long it takes to update the proposed predictor. During the transient period, the current static

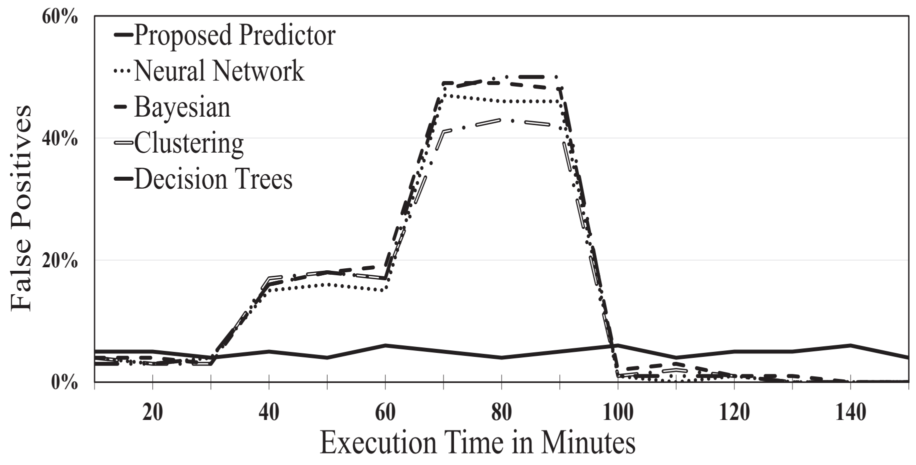

Figure 3.4: FP of proposed vs. current predictors in dynamic environments.

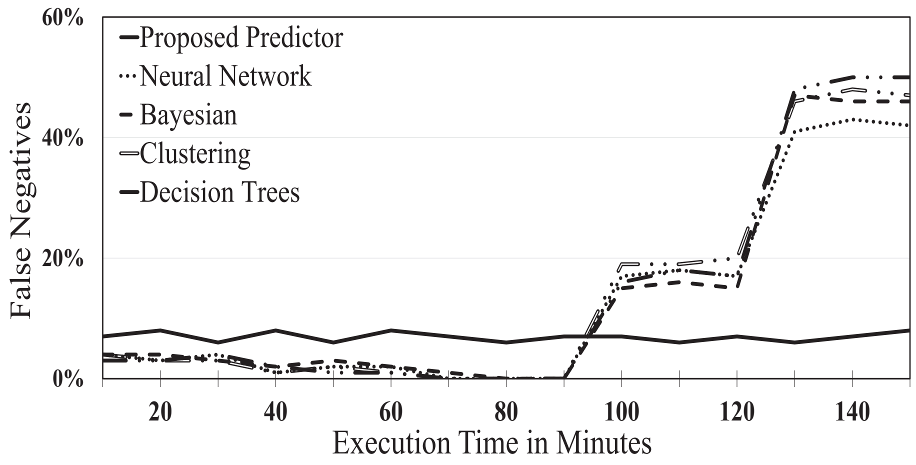

Figure 3.5: FN of proposed vs. current predictors in dynamic environments.

predictors and proposed dynamic predictor drop in accuracy. However, since the proposed predictor was updated after each change, its accuracy during the transient period is far better than the static ones, because their accuracy drop accumulates over time, as shown in Figure 3.3.

3.4.5 Receiver Operating Characteristics of the Predictors

To compare the performance characteristics of the proposed dynamic predictor with the four current static predictors that are created from real production logs we build a Receiver Operating Characteristic (ROC) curve for each predictor. The ROC curves show the relationship between the specificity and sensitivity of the predictors. This is a standard methodology for comparing predictor accuracy over a range of controlled input specificity. The following metrics are used to generate the ROC curves:

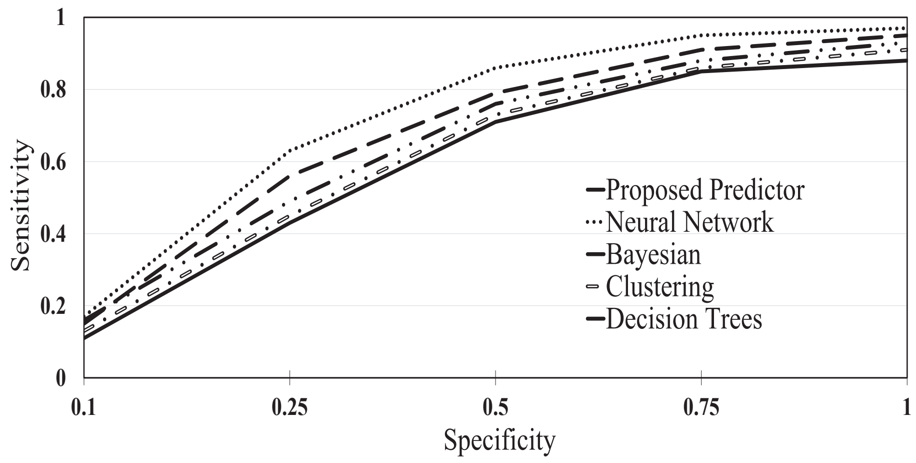

Figure 3.6: Predictors ROC Curves.

- True Positives: when a predictor correctly predicts a transaction to fail to meet the SLA time.
- True Negatives: when a predictor correctly predicts a transaction to meet the SLA time.
- True Negative Rate: is the ratio of true negatives to the sum of true negatives and false positives. This is known as **Specificity**.
- Recall: is the ratio of true positives to the sum of true positives and false negatives. This is known as **Sensitivity**.

Figure 3.6 shows the ROC curves of the predictors in their steady state, when the static predictors are still relevant to the system; note that post system changes, it is no longer possible nor meaningful to plot the ROC curves for the static predictors. The goal of this comparison is to show that the proposed predictor has comparable and viable performance characteristics to industry standard commercial predictors. The main advantage of the proposed predictor is that it takes a fraction of the time to build, train and test in real-time, whereas the static predictors require orders of magnitude more data and time to build, train, and test. Table 3 shows a comparison between the predictors in terms of their data generation and processing times, as well as data Recall.

The downside to the proposed predictor is that it requires high-level knowledge about the system to be implemented as part of its testing whereas the static predictors do not require that knowledge. We argue that this knowledge is already required by the system designers, implementers, and testers who are the intended audience of our suggested approach.

Predictor	Generation Time	Processing Time	Data Recall
Proposed	3-5 Mins	2-4 Mins	100%
Bayesian	5 Weeks	21 Hrs	0.91%
Clustering	6 Weeks	18 Hrs	0.57%
Decision Trees	5 Weeks	19 Hrs	0.68%
Neural Network	5 Weeks	17 Hrs	0.97%

Table 3.3: Processing time and data needed by each predictor.

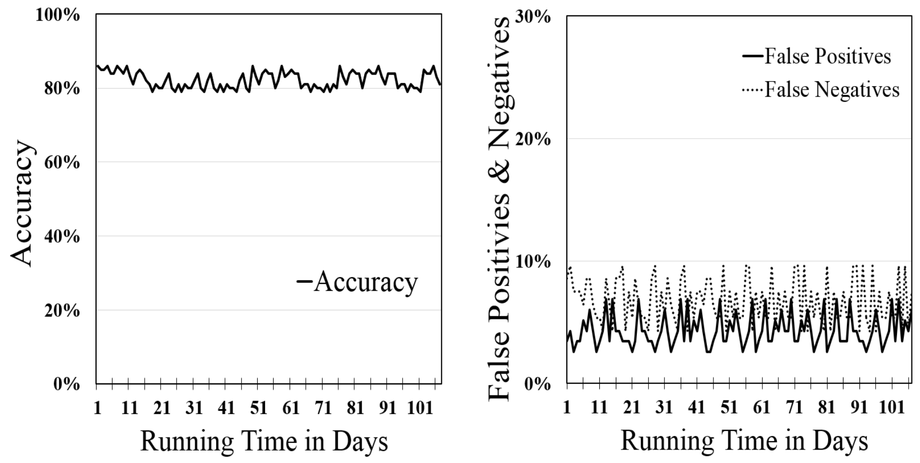

Figure 3.7: Performance of the proposed predictor over 3 months.

3.4.6 Performance Analysis over Long Period

Figure 3.7 shows the daily accuracy, false positives, and false negatives of running the proposed predictor for more than three months. We note the stability and consistent behavior of the predictor over that period of time.

3.4.7 Overheads

The overheads incurred by the proposed predictor fall into two categories: during steady state, and during system changes. During steady state, which is a state where the predictor's accuracy is maintained above 80%, 100 tests are run every minute for an average of 2.5 seconds. Adding 2.5 seconds worth of production testing constitutes less than 5% impact on each production server, which is low. The average real production requests per minute made to each production server is 150 requests. Each request takes an average of 25ms. So the impact on production servers is low. During system changes, when the prediction accuracy starts to fluctuate and drop below 80%, we double the testing load. We run the test every 30 seconds, which results in doubling the amount of time we use the system. We use the system for 5-6 seconds every minute, which is still low. The fact that the LSTs are part of the required production testing, and are used to accomplish other jobs than the prediction maintenance, like testing the actual functionality of the local systems, makes the investment less of an overhead. The test client servers are already dedicated to the production testing functionality, and as such they are not considered an extra cost.

There is no production code overhead special to the predictor functionality. The measurements we take, such as resource utilization and tasks measurements, are provided by the operating system of the production local systems. These are taken at equidistant time series, with or without our approach, as means of monitoring the health of production servers.

3.5 Summary

In this chapter, we presented our first successful effort to utilize synthetic transactions in TiP environments to generate fresh data about the current components of the online service and use that data to build, train, and deploy a real-time failure predictor. The proposed dynamic approach to creating and maintaining failure predictors for online services in real-time showed superior ability to stay relevant and maintain high accuracy throughout the real-time lifecycle of the online service. Using the proposed real-time dynamic failure prediction (RTDFP) algorithm, we can regenerate an online service failure predictor post system changes in a few minutes with a few megabytes of test generated data.

Current approaches to failure prediction are static and cannot keep up with the changes that happen during the real-time lifecycle of online services. Static predictors require massive amounts of data to rebuild, which is not possible in real-time. Static predictors have their strengths and areas of application; they do not require specific knowledge about the system, and are successful in static environments and situations. Online services, however, have dynamic situations that require them to change often.

The proposed approach, requires small amounts of data, in the order of mega bytes (MB), and takes a few minutes to generate, train, test and deploy in production. To evaluate the proposed approached, we deployed it on of Microsoft's display ad services, that is used by millions of users, and receives millions of transactions per second, and we observed enhancements in all aspects of the predictor functionality.

The basics of this effort for generating real time data about the service components, their inputs, system states, and service outputs will be the tool we utilize for our efforts in studying multimedia service capacity estimations and the impact of anomaly detection on multimedia service reliability.

Chapter 4

Enhancing Multimedia Quality by Real-time Service-Capacity Estimation

In the previous chapter, we presented a dynamic approach for monitoring online services, and building a real-time failure predictor. We used synthetic transactions in real-time to generate current service data inputs and captured their induced system states as well as their resulting service outputs. We built correlations between the service inputs, system states, and service outputs that allow us to predict service output failures over time. We were able to keep the failure predictor relevant by updating, training, testing, and deploying it in real-time.

In this chapter, we build on that effort, especially the e2e real-time correlations between service inputs, system states, and service output failures, and extend it with monitoring and measuring the service component maximum capacities that do not result in SLA violations and failures. We focus on multimedia communication services, and extend the mathematical model and correlations introduced in the previous chapter to include the current available capacity of all service components in the dimensional space of the predictive models. By continuously identify the maximum available capacity of the multimedia service components, we create an up-to-date service-wide capacity plan, which enables the service to dynamically construct the multimedia video and audio sessions using to the right combination of service components over the available data centers. This dynamic capacity plan helps maximize the number of active multimedia sessions, maximize the quality of each session, and reduce SLA violations as much as possible.

4.1 Introduction

Interactive multimedia communication services such as Skype, Viber, Whatsapp, and Google Hangout offer a wide variety of services including calling, media sharing, and conferencing services. There are over 2 billion customers who use online communication and media sharing services everyday [42]. A typical interactive multimedia communication service includes client applications and an online cloud infrastructure deployed in multiple data centers around the world, and it uses the Internet as the backbone for communications. When a user calls another user, a client application initiates the session and invokes an online service endpoint in one of the data centers. This end point manages the session by invoking many other sub-services, referred to as components, including identity verification, call management, media adaptation, session routing, experimentation, and advertising. Instances of these components are created and provisioned on different data centers. The perceived quality of session is directly impacted by the performance of these components. Therefore, if we are able to monitor these various components in real-time to assess their current performance and predict any failures before they happen, we can significantly improve the quality observed by users, by routing the sessions and multimedia content through the components and data centers that currently have the highest performance and reliability. As defined in the previous chapter, a failure as an observed deviation of a service component from its expected behavior.

We propose a novel approach to monitoring the health and improving the quality of interactive multimedia communication services **in real-time**. Real-time refers to the runtime lifecycle of the service, where the service is in production and being used by real customers. We use *synthetic transactions* to monitor the service by exercising it like real customers, generate current data about it, and then use this *fresh* data to create and maintain up-to-date failure predictors for the service. The results of the synthetic transactions and the failure predictors are used to make decisions, in real-time, on which services and which paths to route audio and video sessions to.

We implement and evaluate the effectiveness of the proposed approach in a large interactive multimedia communication service in one of Microsoft's Skype data services. The service handles around 2.5 million transactions per second at peak time. We run our experiments for 5 days in a test cluster that gets 1% of the traffic in a data center, and we process more than 100 million audio, video, and conferencing sessions. Our results show that our proactive approach not only substantially improves the quality of interactive multimedia communication services, but it saves network and computing resources. This allows the service to admit more sessions and/or allow current sessions to further improve their quality by adding more streams.

The contributions of this chapter are as follows:

- Novel approach to proactively monitor the health and quality of interactive multimedia communication services.
- Light weight algorithm for building dynamic failure predictors in online multimedia communication services, and using these predictions to estimate the available capacity of different components of the service.

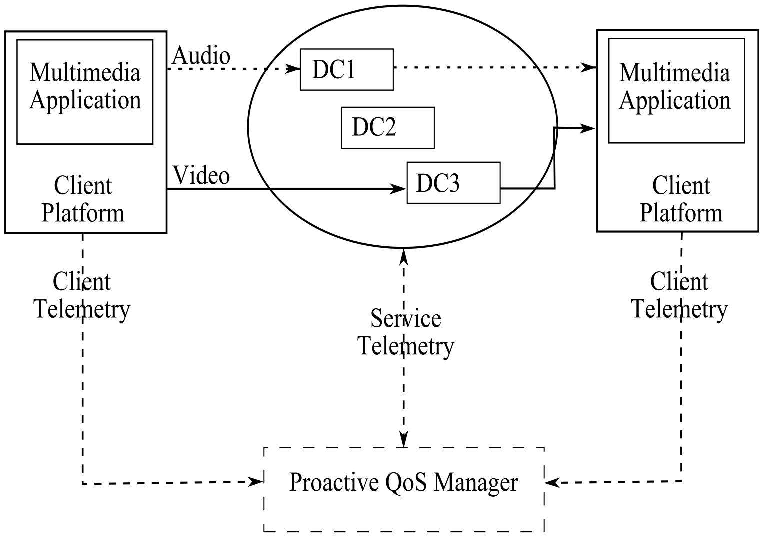

Figure 4.1: High-level architecture of multimedia communication services.

- Algorithm to provide service configuration and routing path recommendations in real-time for the sessions in multimedia communication services.
- Insights and experiences from our actual implementation and deployment of the proposed approach in a large online service.

4.2 Background and Related Work

In this section, we present a high level overview of interactive multimedia communication services, describe how audio and video sessions are created, the points of failure we try to address, and the current research methods that attempt to handle them.

4.2.1 Background

A high level diagram of an interactive multimedia communication service is depicted in Figure 4.1. When a user calls another user, the session is routed to the closest data center. The session may be composed of audio, image, and video streams. Each data center implements an instance of the multimedia communication service that is able to handle all streams in a session. Each data center implements a service routing and selection component. If the data center is able to handle the session, it routes all aspects of the session to its internal components. If the data center is able to handle part of the session, e.g., the audio but not the video stream, it routes the video stream to another data center. Thus, a multimedia communication session may have all its streams going through the same path from the source to the destination, or may have some of the streams of the session go through different paths. An example is shown in Figure 4.1, where the audio from the source client on the left is processed by DC1, and the video is processed by DC3.

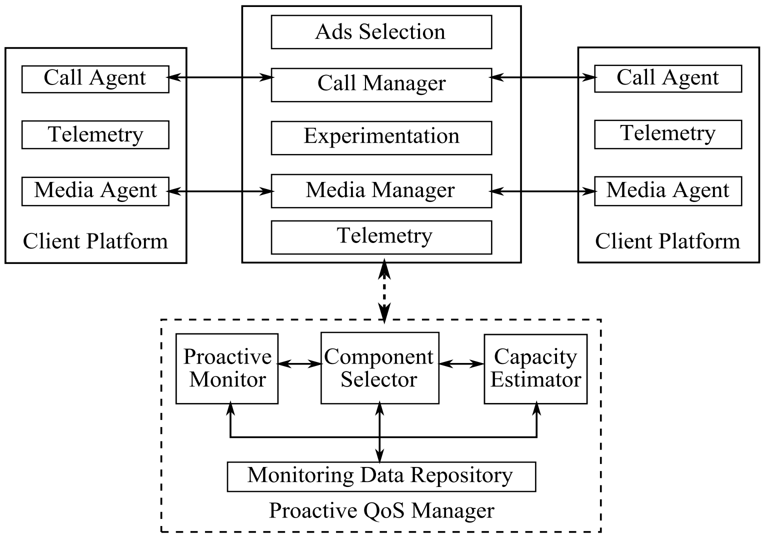

Figure 4.2: Components of MM service and Proactive QoS Manager.

Figure 4.2 provides a more detailed look at the components in each data center as well as the components of the proposed approach, shown in the dotted rectangle. We describe the proposed approach in the next section. For now, we describe the various components in each data center, and how a session is created between two client applications. The service has challenging requirements, and operates under strict SLAs. It receives multimedia communication requests from applications running on mobile devices, Web, and PCs. It processes each request, determines the source device, identifies ads and experimentation configurations based on app categories, device types, user information if available with user permission to use, and user location. It processes the media, and makes a call to the destination client app with the target media, ads, and experimental parameters. Each of the components in the multimedia communication service is comprised of many geo-located instances so that the client call is routed to the most suitable, geo-location and performance wise, data center. The service is composed mainly of Call Manager, Experimentation, Ads Selection, Media Manager, and Telemetry components. The Media Manager estimates the available compute and network resources to maximize the session quality. The Media Manager enables image, audio, and video content to be transported from one endpoint to another as part of a session using UDP or TCP. It implements audio and video encoding and decoding, and packet loss compensation. The Ads Selection component finds relevant ads for the session. The Experimentation component identifies sessions that are suitable for new experiments like new app functionality and/or new color schemes. The Telemetry component collects system performance indicators like available memory and compute resources, as well as functionality measurements like the quality of a video in a multimedia session and number of calls made per minute.

When a user makes a call from an application, the Call Agent tries to connect to the multimedia communication service. The Call Manager of the service handles the incoming session request. It breaks down the multimedia session into its constituents (i.e., audio, video, conferencing, ads, experiments) and tries to identify the best instances in the available data centers that are able to handle the multimedia session. Depending on the used transport protocol (UDP vs TCP) the packets of the session are routed to the selected components, and the connection is established between the two ends of the multimedia session. During the session, the selected components and routing paths do not change.

As can be noted from this high-level description, the quality of the multimedia calling experience has complex dependencies including the selection of components, the routing paths to get to the destination, and the scale and performance characteristics of the Media Manager. There is high demand on the Media Manager component, which may lead the Media Manager to reduce the quality of multimedia streams to ensure that communication sessions do not get dropped. So if we can, in real-time, continuously know the components that have the highest available resources, the highest performance characteristics, and the highest multimedia throughput quality as well as predict how these factors will change over the coming short period of time, we can make component selection and path routing decisions in real-time that will result in high quality multimedia communication sessions.

4.2.2 Related Work

Several works have tried to address the reduced quality stemming from contention in multimedia communication services. For example, Trajkovska et al. [43] propose an algorithm to join P2P and cloud computing to enhance the Quality of Service (QoS) of multimedia streaming systems. They investigate cloud APIs with built-in functions that allow the automatic computation of QoS. This enables negotiating QoS parameters such as bandwidth, jitter and latency. The work in [24] deals with the limitations caused by the platforms where the multimedia streaming takes place. Tasaka et al. [44] study the feasibility of switching between error concealment and frame skipping to enhance the Quality of Experiences (QoE). This approach utilizes a tradeoff between the spatial and temporal quality that is caused by error concealment and frame skipping. The algorithm they suggest switches between error concealment to frame skipping depending on the nature of errors encountered in the video output. Ito et al. [45] study the tradeoff between quality and latency in interactive audio and video applications with the Internet being the medium of communications. They note that the temporal structure of audio-video communication is disturbed by the delay jitter of packets. They study the impact of de-jittering on latency, and the impact of improved latency on quality, which is how the temporal structure is preserved. Through the adoption of psychometric methods, they adjust the initial buffering to enhance latency and quality.

To study the impact of geographical distribution of multimedia services and distributed peers, Rainer and Timmerer [46] suggest a self-organized distributed synchronization method for multimedia content. They adapt IDMS MPEG-DASH to synchronize multimedia playback among the geographically distributed peers. They introduce session management to MPEG-DASH and propose a new distributed control scheme that negotiates a reference for the playback time-stamp among participating peers in the multimedia session. The goal is to improve synchronization, enhance quality, reduce jittering, and enhance latency [47].

A number of works have attempted to address video rendering problems in real-time. Li et al. [48] propose a new rendering technique, LiveRender, that addresses the problems of bandwidth optimization techniques like inter-frame compression and caching. They address problems of latency and quality by introducing compression in graphics streaming. Shi et al. [49] propose a video encoder to select a set of key frames in the video sequence which uses a 3D image warping algorithm to interpolate non-key frames. This approach takes advantage of the run-time graphics rendering contexts to enhance the performance of video encoding.

The aforementioned approaches work on the limitation of the systems, and try to make the best of available resources to enhance the multimedia communication experiences. These schemes aim at controlling QoS, and managing latency and lack of quality. They come short of addressing the actual cause of the problem, which is knowing the state of the services, identifying the services and components that are not performing well, and/or will not perform well in the coming period, and trying to avoid contention and congestion before they happen. We propose a different approach to enhancing the quality in multimedia communication services. We proactively and dynamically monitor the multimedia services to get an insight into their state, and continuously know the best available components where there is minimal contention and congestion, best quality, and least jitter and delay.

4.3 Proactive QoS Manager

We define three terms that we use in this chapter: **Application**, **Service**, and **Platform**. Application refers to the software used by real customers to make calls. Service is the online backend system that applications invoke to accomplish the required functionality such as calling other parties and sharing media with them. A service may be comprised of one or more online services. To avoid confusion, we refer to these other services as components of the main service. Platform refers to the stack of software, hardware, and operating system that is used to run the applications and services. For example an Apple iPhone is a client platform, and Microsoft Azure is a service platform.

4.3.1 Overview

The proposed approach, denoted by Proactive QoS Manager in Figure 4.2, monitors the health and QoS of multimedia communication services and predicts failures in real-time. We use synthetic transactions to proactively and continuously test the services and platforms and generate current data about them. The data is used in its ephemeral state to: (1) provide situational awareness about the whole service, (2) correlate failures between applications, services, and platforms, and (3) build a failure prediction algorithm that predicts future failures. The data is used to keep the failure prediction system up-to-date with the changes of the services and platforms. These three steps result in the ability to make decisions about which components to use that result in the best possible multimedia session experiences.

Before getting into the details of the proposed approach, we describe the Testing in Production (TiP) concept, because our approach uses it. TiP entails software testing techniques that utilize real production environments, while mitigating risks to end users [38, 39, 40]. Online service providers such as Facebook, Google, Microsoft, and Yahoo use TiP to perform functional, stress and performance testing in real-time [40]. Synthetic transactions are used in TiP to generate testing loads, and they are marked with special moniker(s) so that they are distinguished from real transactions. Synthetic Transactions should not interfere with the service destructively, or result in an incorrect state of the service like product inventory reduction due to test purchases. Synthetic transactions in TiP do use the computing resources of the tested service. Service designers account for such an impact due to the importance of TiP; without it the service would be flying blind [38, 40]. We add our synthetic transactions for QoS management to the existing TiP transactions as described in the next section. No code is instrumented in the production code of the service to generate the data for our approach. This is an important advantage for our approach, because we do not impact the production code, and thus no extra testing is needed. Updates to the service monitoring and failure predictor do not require production code update or redeployment.

Figure 4.2 shows the high-level diagram of the multimedia communication service with the proposed approach in the dotted rectangle. The proposed approach consists of four components: (1) Proactive Monitor, (2) Capacity Estimator, (3) Component Selector, and (4) Monitoring Data Repository. The first three components are implemented for each service instance deployed in a single data center, and the Monitoring Data Repository is centralized to all instances; i.e., one instance for the whole system. The Proactive Monitor tests the individual components of the service as well as the whole service e2e. It collects the test results and makes them available to the Capacity Estimator and the Component Selector. The Capacity Estimator uses this data to implement a dynamic failure predictor and to keep it up-to-date to handle any changes. It predicts the capacity of individual components of the service and the loads at which the components will start to violate their SLAs. The Component Selector collects the current status of the service and its components from the Proactive Monitor, and the capacity limits that will result in SLA violations for the coming monitoring period from the Capacity Estimator. It then creates a **Service Capacity Plan** for the coming monitoring period. The plan contains the available components in the service, their current state, and their projected capacity before SLA violations start to happen. It pushes this plan to the Monitoring Data Repository which gets the service capacity plans from all service instances from all data centers. It combines these plans into one **Global Capacity Plan**, and makes it available to each service instance to pull through an API. The plans are updated regularly; the frequency of update depends on the service at hand, typically this would range from 30 seconds to a few minutes. The following subsections describe each of the elements of the proposed approach.

4.3.2 Proactive Monitor

The goal of the Proactive Monitor is to provide real-time insights into the state of the multimedia communication service. We define two concepts: **Local System** and **Scenario**. Local System refers to the stack of software, hardware, and operating system used to perform a specific functionality in the online service. Scenario refers to the collaboration of the set of local systems that are used to implement an e2e user scenario. As an example, assume a multimedia communication service that allows users to invite other users to watch a video together. The e2e scenario is the process of calling others and sharing videos with them. Assume the event of interest is video quality. The video quality SLA for the e2e transaction could be 4, i.e., good quality, using the Mean Opinion Score (MOS) [50]. The video sharing process meets its SLA if the video MOS quality is 4 or 5, and fails otherwise. Assume that the sharing service makes the following calls behind the scene: buffer and encode video, transmit video, receive video, decode video, and de-jitter video. Each of these calls represents a local system. The collaboration of these local systems to implement the video sharing process represents the e2e scenario.

We define three levels of monitoring: (1) local system level, (2) scenario level, and (3) platform level. The Proactive Monitor implements the first level by regularly performing Local System synthetic transaction Tests (LSTs), to continuously monitor the health of each local system. For example, this would be calling the video buffer and encode function and passing it a video, and noting its output video and the time it took to perform the encoding. The Proactive Monitor implements the scenario level of monitoring by performing an e2e Scenario Synthetic Transaction Tests (SSTs) that implement the functionality provided to customers, i.e., sharing a video between two clients. The SST test calls all the local systems that collaborate to perform the functionality, and notes the e2e output of the scenario. These two sets of tests, the LSTs and SSTs, are used to monitor the individual local systems and the e2e scenarios. The Proactive Monitor implements Platform Synthetic Transaction Tests (PSTs) to collect performance indicators like CPU utilization, memory consumption, and process count to correlate with the observed failures. We do not implement application synthetic transactions, i.e., test calls to the applications, as applications are generally installed on user devices and are mostly on metered networks. We use the default client telemetry implemented on these devices that provide information about the application usage. We combine this information with the service and platform synthetic transaction information to monitor the service e2e, and to generate failure prediction systems.

The Proactive Monitor makes LST and SST calls using test loads, which are similar to real loads. The real user behavior, loads and call distributions, is found from logs of previous deployments of the service, or from field/market studies about the service. For example, it would be known, based on previous production logs and/or estimated from market research, that the average user of a multimedia communication service shares a video of 60 seconds with 2-5 friends at a time, and that the service gets around 1,000 concurrent users at peak times. The SSTs made by the Proactive Monitor start with these loads, and progressively add more loads until the failure causing loads are found. If the SLA of video quality is, say, a minimum MOS value of 4, then any video quality of MOS value less than 4 is considered a failure. The test loads are executed on the current service, thus they generate information that represent the current state of the service and its performance characteristics. The synthetic transactions are run for a short period of time, e.g., 5 seconds every couple of minutes. Thus, they do not increase the load on the service considerably.

The LST, SST, and PST calls are made at equidistant time series; once every M seconds. The value of M is configurable, depending on the service, and it varies during run time, depending on the state of the service. If the service is churning and failures are happening more, then M is reduced to get a better pulse of the service. The event, i.e., video quality, is captured after each of these tests, and its value is compared to the result of the prediction. If the accuracy of the predictor starts to drop, the tests are done at a higher rate to generate data to build a new predictor.

The set of tests performed every M seconds need not be exactly like those of customer behavior. However, if the service at hand requires an exact replica of the user behavior, then parts of previous production logs representing real user behavior can be replayed as suggested by [37, 16, 26].

4.3.3 Capacity Estimator

The Capacity Estimator tries to find the maximum capacity that the service and its components can handle before SLA violations start to happen. It does so by predicting the capacity at which failure starts to happen. It implements a dynamic failure predictor that is able to cope with the service and component changes in real-time. The failure predictor is an independent module that runs in the same environment where the online multimedia service runs. It is part of the TiP system. It has access to the same resources as the production service, but is not part of the online service code. Updates and re-deployments of the predictor code can be done anytime without interfering with the service production system.

Failure predictors are designed to predict the outcome of an event [33, 34]. In multimedia communication applications and services, events represent a variety of measurable aspects of the service, such as the call quality, the response time of the service, and the availability of the service. In this chapter, failure prediction means predicting when the outcome of an event does not meet its SLA. For example, if the event of interest is multimedia quality, then failure prediction means predicting the cases where the quality of the shared media drops below the MOS value specified in the SLA. We need a dynamic failure predictor that can be created and updated in real-time. We implement one of the latest failure predictors called Real-Time Dynamic Failure Prediction (RTDFP) algorithm [3]; other predictors can be used in our approach as well. We choose RTDFP because it can be created and maintained in real-time in a short amount of time, and has high prediction accuracy. We customize RTDFP for our Proactive QoS Management approach. The main steps of the Capacity Estimator are summarized as follows:

- Step 1: Use the Proactive Monitor to exercise the service and generate current data about it.
- Step 2: Collect, from the Proactive Monitor, the synthetic transactions and applications' inputs, service and platform outputs, e2e scenario outputs, such as media quality and response times, and events of interest. This data is collected into in-memory constructs with a predefined dimensional model (explained below).
- Step 3: Build a real-time predictor, from the data collected from the applications, services, and platforms and the events of interest.
- Step 4: Estimate the available capacity based on the outcomes from the failure predictor.

The data collected in Step 2 includes the current load and state of the service like the number of videos to process, and the used resources like media processors and call dejitters. Service and platform related data such as memory utilization, number of processes, and CPU utilization are collected. We describe in the evaluation section how we capture such data in our experiments. The data is collected and hosted in memory in an array with a dimensional model schema [35]. Dimensional modeling refers to a set of techniques used in data analysis to provide insights into the cause-effect relationships between entities. Data is organized into two sets, a set of measured or monitored data, called **facts**, and a set of parameters, called **dimensions** that define and control the facts. For this chapter, the fact is the event of interest that is to be predicted, such as media quality. The dimensions are the other sets of data that impact the event, such as:

- The **LST** that is called: this allows us to know which test is run.
- The **time** that LST is called: the time stamp allows us to relate the cause and effect in the service calls, i.e., at a specific time there were that many function calls and tasks, which caused the observed response time.

Dimensions					Fact
LST	Tasks	Data center	Component	LST time	Video Quality (MOS)
1	53	3	5	23	4
2	53	3	2	17	4
3	53	3	3	31	3
4	53	3	1	19	5

Table 4.1: Dimensional Model for the online Multimedia Service.

- The **number of tasks**: the number of media processing jobs in the service.
- The **component**: the specific component, like Ads Selection or Experimentation, in the service instance.
- The **data center**: where the components are deployed; this could include more details such as the deployment ID.

Table 4.1 represents a service QoS dimensional model. Note that we use surrogate keys [35] to represent non-measured and non-numeric values. A surrogate key is a unique identifier of an entity; it is an integer and it is not derived. This makes a table with smaller footprint in memory, as only integers are used, and enhances the performance of operations done on it. The map between the surrogate keys and the actual values is stored in a local table that has the actual dimension values and the corresponding surrogate keys. For example, the service instance in the US western data center maps to "Dim3 Service" surrogate key value 3.

The collected data is used for two purposes. The first purpose is to feed the Monitoring Data Repository to build reports and dashboards that represent the current state of the service such as how many users are making multimedia calls in a given data center and the average quality of shared media over time. This provides situational awareness about the service on an ongoing basis. The other purpose that we use this data for is to generate a real-time failure predictor, and maintain its accuracy over time (Step 3).

A predictor is a classification system [33, 34] that defines and monitors boundaries of working conditions that result in an event success or failure. The event could be the MOS quality. The working conditions are the independent and dependent variables that control the outcome of the event. We use the load of each component as the independent variable, and the CPU utilization as the dependent variable; because it is dependent on the load of each component, yet it plays a key role in controlling the event outcome. We implement a light-weight predictor based on RTDFP that uses the load and the CPU utilization of each component as a logical expression model [3]. The logical expression model is a set of conditions defining the safe component working-conditions that result in the success of the event. We define the following model for the MOS value to be successful:

- MOS Quality Condition for Success for Component N:
  - Independent Variable for Component $N <$ $IndependentVariableMaxValue$ found by the Proactive Monitor
  - Dependent Variable For Component $N <$ $DependentVariableMaxValue$ found by the Proactive Monitor

The Capacity Estimator (Step 4) gets the maximum values for the independent and dependent variables that are found by the Proactive Monitor LST and SST tests, builds the Service Capacity Plan for the coming monitoring period, and passes it to the Component Selector. In other words, the Capacity Estimator utilizes the RTDFP failure prediction to build the component capacity for the coming monitoring period that will not result in SLA violations.

4.3.4 Component Selector

The Component Selector combines the current data about the service from the Proactive Monitor with the projected capacity that the service and its components can handle from the Capacity Estimator. Table 4.2 shows an example of this combined data that represents the Service Capacity Plan for the coming monitoring period. The Component Selector pushes Table 4.2 to the Monitoring Data Repository to be combined with similar data about the components of the rest of the data centers. The Component Selector pulls the Global Capacity Plan from the Monitoring Data Repository into a new instance of Table 4.2.

The Service and Global Capacity Plans are computed at the beginning of each monitoring period. Each Component Selector reads these plans from the Monitoring Data Repository, and makes them available for its local service instance in its data center. The local service reads the Global Capacity Plan from the Component Selector into an in-memory array to make access to it fast and suitable for routing every session in real-time. When a client invokes the local service, the local service finds the most suitable component to handle the session from its in-memory copy of Table 4.2, and routes the session to it. Two factors are used in determining the most suitable component: (1) geo-graphical location; the Data Center dimension in Table 4.2 is used to find the closest component that can be used, so locally first then the same region, and (2) the component that has the highest available capacity; the Projected Extra Media Jobs fact in Table 4.2 provides this information. Once a session is committed to a component, it remains there until completed. In other words, session routing to components using the Global Capacity Plan affects new sessions only. This prevents session oscillation between components.

As shown in Procedure 1, the Component Selector implements three functions: GetCurrentStateOfServices(), GenerateServiceCapacityPlan(), and GetGlobalCapacityPlan(). The function GetCurrentStateOfServices() gets the current loads and status of each component

Center	Component	Processing Jobs	Avg MOS	Projected Media Jobs
4	5	36	5	28
2	6	41	5	23
3	2	27	5	19
7	4	17	5	17

Table 4.2: Service Capacity Plan.

in the system from the Proactive Monitor. This function returns an instance of Table 4.1. GenerateServiceCapacityPlan() uses the failure predictor to predict the loads that will result in SLA violations. These loads are used to determine the available capacities in the components for the next monitoring period. The available capacities are combined with the instance of Table 4.1 to produce an instance of Table 4.2, which is the Service Capacity Plan. GetGlobalCapacityPlan() pushes the Service Capacity Plan to the Monitoring Data Repository which gets all such plans from all Component Selectors. The Monitoring Data Repository combines all these instances into the Global Capacity Plan. GetGlobalCapacityPlan() pulls the Global Capacity Plan from the Monitoring Data Repository and pushes it to the service configurations so that the service can use it in routing sessions.

4.3.5 Monitoring Data Repository

The main function of the Monitoring Data Repository is to combine the local Service Capacity Plans into one Global Capacity Plan that represents the whole system, and to make it available to all components to pull as needed. This is an important functionality that prevents excessive communication between the Component Selectors of each data center instance of the Proactive QoS Manager. With the Monitoring Data Repository, every Component Selector computes the Service Capacity Plan, and communicates twice with the Monitoring Data Repository. The first is to send its Service Capacity Plan, and the second is to get the Global Capacity Plan. The Monitoring Data Repository gets all Capacity Plans, merges them, and generates one Global Capacity Plan that represents the current usage of all components as well as their projected capacities. The Monitoring Data Repository is implemented as a single instance. It is part of the TiP system, so it does not have the high availability requirements that production systems have. In case it is down, each Component Selector continues to use its own Service Capacity Plan until the Global Capacity Plan is built. The production service uses the suggested Proactive QoS Monitor as a heuristic agent to enhance its component selection and routing functionality. The production system is designed to survive and function well, even if the whole TiP system is down.

Procedure 2 Component Selector Algorithm

```
COMPONENT SELECTION
    function GetCurrentStateOfServices
    Select Service, Component, ProcessingJobCount, AvgMOS
            from Table 4.1
            orderdescendingby AvgMOS
            groupby Service and Component;
    end function

    function GenerateServiceCapacityPlan
    GetCurrentStateOfServices();
    Use RTDFP to get Predicted JobCount until failure;

    MergeJoin Ordered list with Predicted JobCount until failure
                orderdescendingby Service and Component;

    Select Service, Component, ProcessingJogCount,
            AvgMOS, JobCountUntilFailure
            orderdescendingby AvgMOS and JobCountUntilFailure;

    Populate Service Capacity Plan instance of Table 4.2;
    end function

    function GetGlobalCapacityPlan
    Push Service Capacity Plan to Monitoring Data Repository;
    Wait until Monitoring Data Repository builds Global Capacity Plan;
    Pull Global Capacity Plan from Monitoring Data Repository;
    Push Global Capacity Plan to local service Configurations component;
    end function
```

4.3.6 Evaluation

We implement the proposed approach in an operational online multimedia communication service offered by Microsoft [51] and we present results from more than 100 million video and audio sessions.

4.3.7 Implementation and Setup

We implement LSTs for four local systems: Call Manager, Experimentation, Ads Selection, and Media Manager. The geo-distributed multimedia communication service processes an average of 2.5 million requests per second at the peak. It is deployed in 8 data centers in 3 continents. LSTs are API calls to each of these components, with loads as explained in the Proactive Monitor section. The setup we implement is based on the model shown in Figure 4.2. We use a small test cluster of 10 servers in the data center, which gets around 1% of the data center traffic to run our experiments. Each server is a quad-core intel Xeon server with 12 GB RAM. We implement a Proactive Monitor that makes LST, SST, and PST calls to exercise each component and we capture the inputs to the LSTs and the outputs of the Local systems. Similarly, we record the output of SSTs and PSTs. An SST is composed of the LSTs that represent the scenario; so a session with audio and video is composed of Call Manager, Experimentation, Ads Selection, and Media Manager LSTs with parameters that specify the audio and video characteristics like length and encoding. We measure the SST response time and quality of the multimedia session using a proprietary automated MOS algorithm. PSTs are implemented in an infinite loop that reads CPU utilization, memory utilization, and number of processes from the performance monitoring APIs of the operating system of each component every 30 seconds.

The Proactive Monitor makes the calls to each of the local systems and controls the various aspects of the multimedia communication request, including media type, media size, location, and automated user information. The Proactive Monitor simulates multimedia sessions made by multiple clients, by making simultaneous calls with different user agent information. It controls the load in two ways: the number of media sessions made by each simulated client in a given time, and the number of simultaneous media sessions representing multiple client calls. The data is collected into in-memory arrays with a schema similar to Table 4.1. The synthetic transactions are designed to run 100 concurrent LSTs every minute. The accuracy of the predictor is measured by comparing its predictions with the actual results and monitored event value from running LSTs and SSTs. Each LST set of tests takes an average of 5 seconds to complete. The resulting data from the Proactive Monitor synthetic transactions are collected by the Capacity Estimator and Component Selector. The Capacity Estimator makes capacity estimates for the coming monitoring period of one minute. The Component Selector builds the Service Capacity Plan every minute, and communicates with the Monitoring Data Repository to get the updated Global Capacity Plan.

4.3.8 Performance Metrics

We study the quality of media during the multimedia session and the number of shared media streams, i.e., video, audio, and image. We use the following metrics in our experiments:

- **Media Quality**: the quality of media (audio, video, and image) that is shared between clients. We use MOS for this metric. We use a proprietary *automated* MOS algorithm. Other automated MOS test software is commercially available, e.g., [52]. These automated MOS algorithms enable quality measurements in test environments, where sessions are generated programmatically between test clients.

- **Number of Failures**: the number of sessions that failed to meet the required MOS quality as specified by the service SLAs.

- **Increase in Service Capacity**: the number of additional sessions that the service can handle. This value is found by the Capacity Estimator and using the Proactive Monitor tests to verify its accuracy. We measure the service capacity, by finding the loads that the service can handle until failure, with and without the proposed approach.

- **CPU Utilization**: this reflects how much of the total available resources are freed up due to better resource utilization and load balancing. We get this as part of running the PSTs, through an operating system API call every 30 seconds. This is extracted for each component.

- **Overhead**: We measure the CPU utilization of the service with and without the TiP system, to find the overhead of TiP.

4.3.9 Results

The service we test our approach in has a high track record for meeting its strict SLAs; on average and over a period of almost three years, the service is able to meet its multimedia communication quality SLA more than 99.9% of the time. The service strives to do better, as a 0.1% failure rate is still high and costs a lot of money for a service that manages 2.5 million transactions per second at peak. That's 25,000 transactions per second at peak that potentially did not meet SLA. To provide an idea of the monetary impact alone of such a failure rate, note that the ads monetization for such a service runs at a rate higher than \$10 per a thousand ad impressions. That is, advertisers pay \$10 for every 1,000 ads shown to customers. So that's an opportunity loss of minimally \$250 *per second*, let alone the

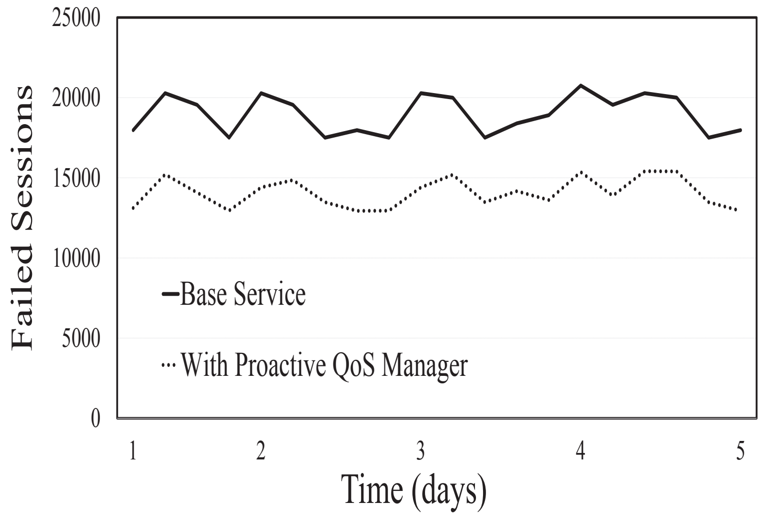

Figure 4.3: Failure reduction with proposed approach.

negative impact of the lower customer satisfaction, which can lead to losing customers to competitors.

Our test cluster gets around 3,000 transactions per second at peak. We ran the experiment for 5 days, and we divide the test time into two alternating parts: (1) the base or reference part, where we run the traffic over the service without the proposed solution for one hour, and (2) the updated service or optimized part, where we run the traffic over the service with the proposed solution for another hour. We aggregate the results of our tests at a granularity of 6 hours; i.e., each point in the graphs we show in this section represents a 6-hour aggregation, unless otherwise mentioned. In each 6-hour period, 3 hours have the results for the base service, and the other 3 hours have the results for the optimized service with our Proactive QoS monitoring approach.

We processed more than 100 million sessions over the 5-day period. For each period of 6 hours, the average number of multimedia sessions we get is around 5 million, half of them with our approach and the other half without it.

Number of Failures: We measure the number of failed sessions with and without our approach, and we plot the results in Figure 4.3. As the figure shows, the average number of failed sessions is around 19,000 failures without our approach. Using the proposed approach to monitor, estimate capacity, predict failures, and route to components with higher capacity and better performance, the failed sessions count drops to an average of 14,000. Therefore, the proposed approach manages to drop the failed sessions by an average of 26%. The average is fairly smooth because it is measured across a large period of 3 hours. The gain from our approach is much higher during peak times, where resources are constrained.

Media Quality: We measure the media quality of the succeeded sessions. Our results show that not only did the failed session count drop by an average of 26%, but the quality

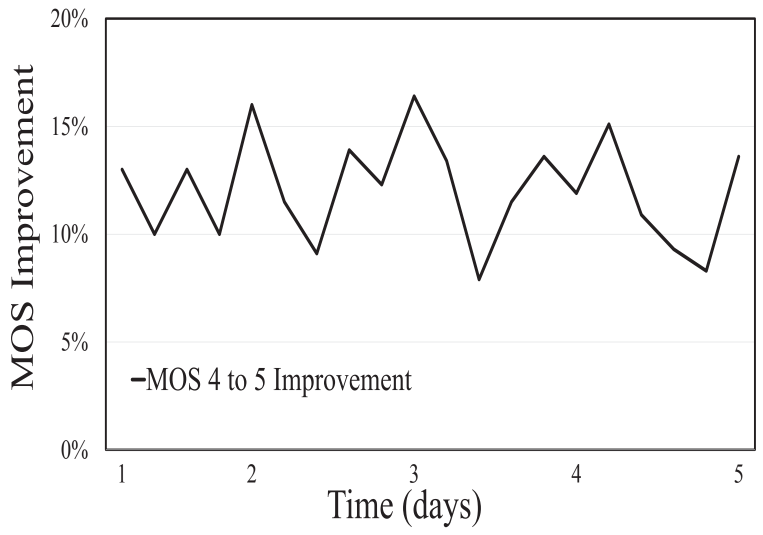

Figure 4.4: MOS increase from 4 to 5 over 5 days.

of the sessions that meet SLA have seen an improvement as well. Figure 4.4 shows that, on average, around 12% of the sessions that used to meet SLA with MOS 4, are now meeting their SLA with MOS 5, Excellent Quality. We study the MOS improvement from 4 to 5 in more details over 12 hours, while alternating between using the service with and without our approach every hour. We average the MOS measurement every 5 minutes. We see an improvement of about the same average, around 12% as shown in Figure 4.5, that we see in the 5 days experiment with 6 hour aggregation. This shows the stability and predictability of the proposed approach over different time ranges and aggregation periods.

Increase in Service Capacity: The average available service capacity has seen an improvement. This is the number of extra multimedia sessions the service can handle. Because of the enhanced component selection, the service is seeing less bottlenecks and delays, and so is able to handle more multimedia sessions. Figure 4.6 shows that the service has an average of 17% session capacity increase over the 5 day experiment, and a maximum of up to 21% increase can be achieved.

CPU Utilization: The service CPU utilization has seen a drop of an average of 10% over the 5 day experiment, as shown in Figure 4.7. This is closely related to the increase in service capacity. Because of the optimized component selection, the components are observing better load distribution and so less congestion. The demand on their CPUs has dropped by an average of 10%, which allows the components to handle more sessions, by an average of 17%, with the same resources.

Overhead: we measure the overhead of the TiP system on the production system to determine the cost of the proposed approach. We measure the service CPU utilization with and without TiP. We find that the average service CPU impact caused by the whole TiP system is around 3% over 5 days, as seen in Figure 4.8. This includes the proposed

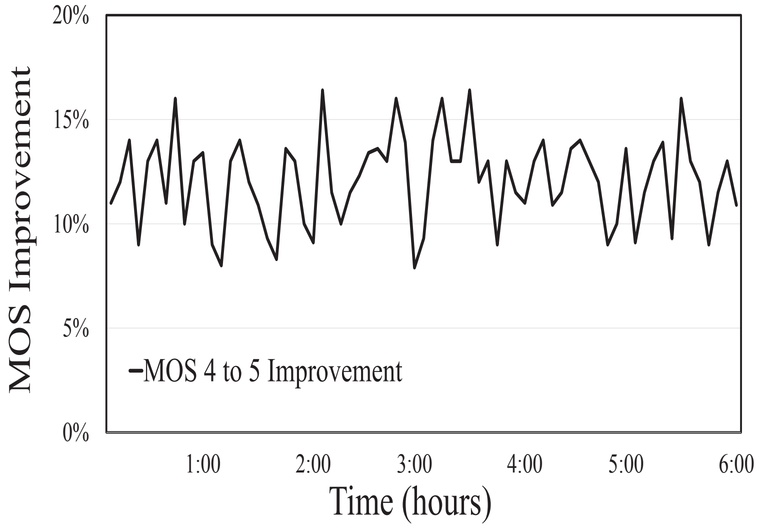

Figure 4.5: MOS increase from 4 to 5 over 6 hours.

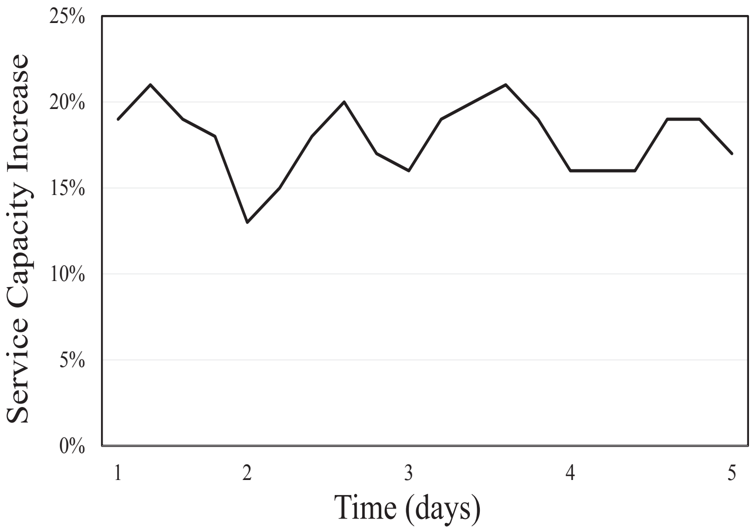

Figure 4.6: Increase in the available capacity.

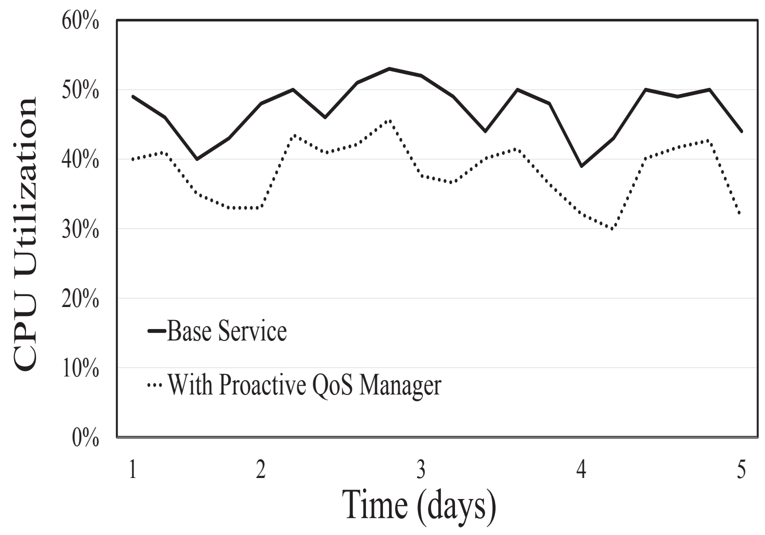

Figure 4.7: Reduction of CPU utilization.

Proactive QoS Manager approach as well as all other testing in production functionality. All computations of our approach are done on the TiP system, not the production system. The improved multimedia quality, reduced failures, enhanced service session capacity, and reduced CPU usage are gains made by the proposed approach that make the overall investment in the TiP system more than justified.

4.4 Summary

In this chapter, we presented a real-time approach to determine the maximum available capacity of online multimedia service components before they start to fail and impact customers. We dynamically monitored the health and quality of multimedia communication services. We used *synthetic transactions* to monitor the service by exercising it like real customers, generated current data about it, and then used that *fresh* data to create and maintain up-to-date capacity plans and predictions of different components of the multimedia communication service. We then made component selection and routing path actions in real-time to enhance the quality of multimedia sessions.

We evaluated the proposed approach in a production system, in one of Microsoft's Skype data services, and on average, the proposed approach increased the overall media sharing quality by 12%, decreased the percentage of failures by 25%, reduced the CPU usage by 10%, and increased the session capacity in the service by 17%.

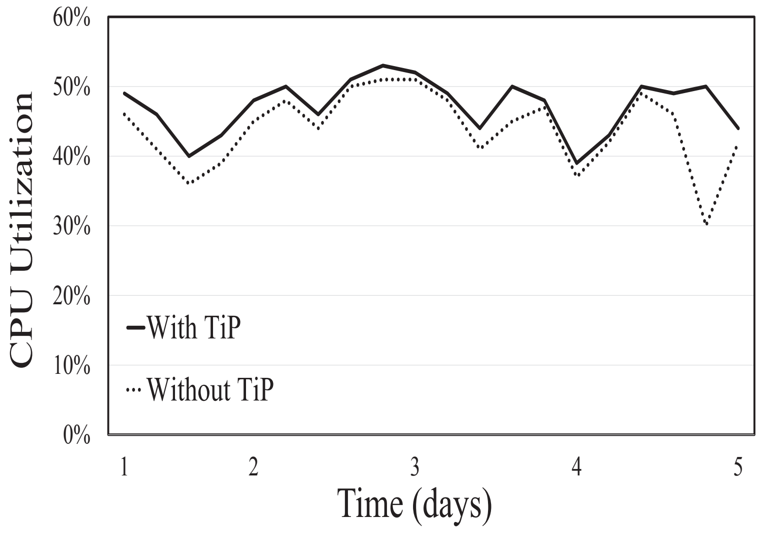

Figure 4.8: TiP impact on CPU utilization.

Chapter 5

Dynamic Anomaly Detection in Interactive Multimedia Services

In the last two chapters, we built real-time failure predictors for online services, and capacity and traffic management plans by finding the maximum available capacities in multimedia service components that would not result in SLA violations and failures. The next step for us, in the continuum of understanding the impact of service inputs and system states on the service outputs and SLA violations, is the study of the impact of anomalies on multimedia service outputs and SLAs. An anomaly refers to values of the service inputs, component working conditions, and system states that deviate from their expected range [17].

In this chapter, we attempt to understand the impact of anomalies in the multimedia service inputs, working conditions, and system states on the service outputs. This allows us to manage anomalies, improve the reliability and performance of multimedia services, and reduce the observed SLA violations and failures. Working conditions refer to the activities running on the service like number of active processes and number of open sockets. System states refer to the service internal states like CPU and memory utilization.

5.1 Introduction

State of the art anomaly detectors follow a data-driven and mathematical models to define an anomaly [17, 18]. They analyze large amounts of historic data that represent the service inputs and its working conditions, such as number of users and their active video sessions, and the number of active processes and their CPU utilization. Using mathematical modeling, current anomaly detectors find the expected values for these properties as well as the outliers or anomalies [17, 41]. In the current approaches for handling anomalies, service implementers generally raise alerts and request corrective measures when anomalies are encountered, without considering the actual impact of anomalies on the service [3, 14]. Alerts are expensive from a system and engineering support perspectives, and should be raised only if necessary [43].

There are multiple key problems with this common theme in anomaly detection and handling approaches, especially for interactive multimedia services. First, the use of historic data to determine the impact of anomalies on the service, raise alerts, and request corrective measures. In the case of multimedia services, the data comes from logs of prior runs of the service [9]. That data may no longer reflect the current conditions of the service accurately, as new services are continuously added to data centers, and storage and compute provisions are updated through collaboration with other data centers. For example, if a multimedia service expects no more than 100 participants in an online meeting, then having 110 participants is considered an anomaly that raises alerts, even if the service has enough current capacity to handle the extra participants; so why raise alerts and take the cost hit if that is not needed.

Another problem with state of the art anomaly detection and handling techniques is that they are generally designed to monitor and alert on specific metrics of the system independently; for example, number of processes or CPU utilization [3]. However, it is rarely the case that a single metric can be correlated to multimedia SLA violations. Usually, the output of the multimedia service, like multimedia quality, reduces below an acceptable value under a few conditions together, like number of customers, number of concurrent video sessions, and CPU utilization. So it is important to consider the *association of metrics* that cause SLA violations.

Lastly, the cost of pre-processing previous logs to prepare them for anomaly detection and impact analysis is substantial. The majority of multimedia transactions in production service logs have the expected service inputs and working conditions, and succeed without causing SLA violations [8, 27, 34]. Thus, the SLA violation rate is low, and so is the recall in the data. Recall in this context refers to the percentage of data that is relevant and usable in the analysis of anomalies. Having relevant and current data with high recall is more important to the success of anomaly detection than advanced and deep algorithms [18]. Because of that, hundreds of gigabytes of log data over months are needed to find enough relevant data for anomaly detection and analysis for multimedia services [27]. Using historic data in generating analytical systems like data mining and predictive modeling works well in environments that do not change often; such as transportation systems like airplanes and ships. On the other hand, multimedia services lack such stability over time at many levels including service hardware provisions. The ever-changing landscape of multimedia services, coupled with requirements such as continuous up-times, make the use of historic data about the service challenging and ineffective [3].

We propose a dynamic approach to the analysis of multimedia service anomalies. We use synthetic transactions, explained in Section 3, to generate fresh and small, yet highly relevant data about the current state of the service in near real-time. We employ machine learning techniques to correlate the ranges of anomalous service inputs and its working conditions with the service SLA violations. The anomalies themselves are found using current state of the art techniques. Our proposed approach *identifies the impact of these anomalies, independently and associatively*, on the service and its ability to adhere to SLAs, and recommends whether to raise alerts and invoke corrective measures. We consider service inputs and working conditions anomalies worth alerting on *if and only if they impact the output of the service negatively and result in SLA violations*.

We implemented the proposed approach in one of Microsoft's Skype data services in the application and services group, which handles millions of multimedia sessions per second. We show that the proposed approach is able to reduce the number of false positives in anomaly alerts by an average of 71%, reduce false negatives by 69%, enhance the accuracy of anomaly detection by 31%, and enhance the media sharing quality by 14%. The recall in the data generated by the synthetic transactions is 100%. In contrast, the recall in the production logs is less than 2%. In addition, we show that we can update the anomaly detector in near real-time in around 7 minutes. On the other hand, building a detector model using current anomaly detection techniques for the same service by using production logs requires around 7 weeks of production log data that takes several hours of pre-processing before the data is usable.

The contributions of this chapter are (1) new approach for dynamic anomaly detection in multimedia services in real-time, (2) machine learning method to correlate the multimedia service inputs, working conditions, and system states with its outputs and their SLAs, and (3) actual implementation and evaluation of the proposed approach in a real multimedia communication service.

5.2 Related Work

Multimedia services are subject to conditions that impact their SLAs like data center faults and anomalies in the service inputs and working conditions. The complexity and number of components the service depends on exacerbate the impact of anomalies in any component. Current approaches to anomaly detection range from mathematical and data driven machine learning approaches to system-based methodologies. Chandola et al. [17] present a survey of the available anomaly detection techniques and their applications.

Anomaly detection based on machine learning techniques use either historic data about the system at hand, or rule-based approaches. The output of the current anomaly detection techniques used in online multimedia services is in the form of a set of static boundary conditions on the service inputs and its system states [14]. Outliers in such models are considered anomalies even if they do not result in any SLA violations or failures like dropped sessions. The key issue with almost all machine learning approaches is their dependence on large amounts of data to create, train, and test new models [18, 17]. The data preparation time is too high for real-time changes and updates [3]. In addition, these approaches rely on service production logs to find the ranges of service inputs and system working conditions to identify the outlier boundaries and their impact on the service. Production Logs are complex and hard to mine [8, 9]. Data in logs may not be sufficient for mining, analysis, and anomaly detection models [27, 4]. Pre-processing the logs to prepare them for anomaly detection models is hard and expensive [30]. Leners et al. [28] use service informers to improve the availability of distributed services; these are built using system messages found in production logs from prior runs of the service, not from the current service in real-time. The resulting anomaly detection models created using logs from prior runs of services may not accurately represent the current service [6, 32].

Even with online system-monitoring-based approaches, like Hystrix of Netflix [7], the monitoring is still reactive, as the anomalies, faults, and failures need to happen and customers endure them before they are controlled. Anomaly can manifest in many forms, including in the process of synchronization of audio and video sessions. To study the impact of geographical distribution of multimedia services and distributed peers, Rainer and Timmerer [46] suggest a self-organized distributed synchronization method for multimedia content. They adapt IDMS MPEG-DASH to synchronize multimedia playback among the geographically distributed peers. They introduce session management to MPEG-DASH and propose a new distributed control scheme that negotiates a reference for the playback time-stamp among participating peers in the multimedia session [53, 54]. The goal is to avoid synchronization and latency anomalies, enhance quality, and reduce jittering. Other efforts attempted to provide DASH-based approaches to optimize video streaming [55, 56]. Trajkovska et al. [43] propose an algorithm to join P2P and cloud computing to enhance the Quality of Service (QoS) of multimedia streaming systems. They investigate cloud APIs with built-in functions that allow the automatic computation of QoS. This enables negotiating QoS parameters such as bandwidth, jitter and latency, and avoid wrong characterization of state anomalies.

Many efforts attempted to study the impact of anomalies in video rendering in real-time. Li et al. [48] propose a new rendering technique, LiveRender, that addresses the problems of bandwidth optimization techniques like inter-frame compression and caching. They address problems of latency and quality by introducing compression in graphics streaming. Shi et al. [49] propose a video encoder to select a set of key frames in the video sequence. It uses a 3D image warping algorithm to interpolate non-key frames that otherwise can increase anomalies detected in the frames without true impact on the video session. This approach takes advantage of the run-time graphics rendering contexts to enhance the performance of video encoding. Tasaka et al. [44] study the feasibility of switching between error concealment and frame skipping to enhance the Quality of Experiences (QoE). This approach utilizes a tradeoff between the spatial and temporal quality that is caused by error concealment and frame skipping. The algorithm they suggest switches between error concealment and frame skipping depending on the nature of errors encountered in the video output; this technique avoids characterizing unimportant frames as anomalies.

Our approach is different in multiple aspects. We focus on anomalies in service inputs and system states that result in SLA violations. We find SLA violations as they happen under synthetic transactions, not through failure prediction which can be inaccurate and result in over and/or under-stating the anomalies in the system. We use machine learning combined with multidimensional analysis to correlate the service anomalous inputs and working conditions with the service outputs and SLA violations, and find the individual and associative impacts of these parameters on the output of the service. We change the reaction to anomalies based on their current impact on the service, not their historic impact. We monitor the accuracy of the anomaly analysis model and regenerate it in near real-time if it drops below acceptable accuracy.

5.3 Dynamic Anomaly Analysis

5.3.1 Overview

Multimedia services have complex inter-dependencies between their systems, like call managers, encoders, de-jitters, renderers, decoders, and storage systems [46, 48]. The range of issues that impact the quality of a multimedia session include computation capacity, network bandwidth, as well as the varying customer load on the system. So it is almost impossible to correlate an anomaly of one aspect of the system like CPU utilization, memory consumption, or user count with degradation of the multimedia service, like poor media quality measured by MOS [44, 49]. The anomalies' impact on the service is not consistent throughout the day and the lifecycle of the service; for example, an anomaly of higher number of videos shared per session may result in SLA violations during peak hours, but may have no impact at all during slow traffic. So it is not optimal to have the same reaction to anomalies like raising alerts all the time, as done in the current anomaly detectors, because alerts are expensive, and their impact is not constant.

We propose a novel approach, called Dynamic Anomaly Analysis (DAA), to analyze, in real-time, the anomaly impact of the service input and working conditions that are found in the trailing 30 days, on the multimedia service. We study the anomalies independently and associatively, and classify their impact into three categories (1) Impactful, (2) Borderline, and (3) Non-impactful. Impactful anomalies result in SLA violations. These warrant alerting and managing as we describe later. Borderline are anomalies that do not result in SLA violations, but impact the measure of interest in the service. DAA monitors borderline anomalies closely without raising alerts, until they start to cause SLA violations. Non-impactful anomalies do not have any negative impact on the measures of interest. For example, if the expected shared videos in a multimedia session is one video per session, then sharing two videos side by side is an anomaly. However, if sharing two videos side by side doesn't impact any measure of interest like media session quality, then this anomaly is considered non-impactful, and DAA takes no actions on these anomalies.

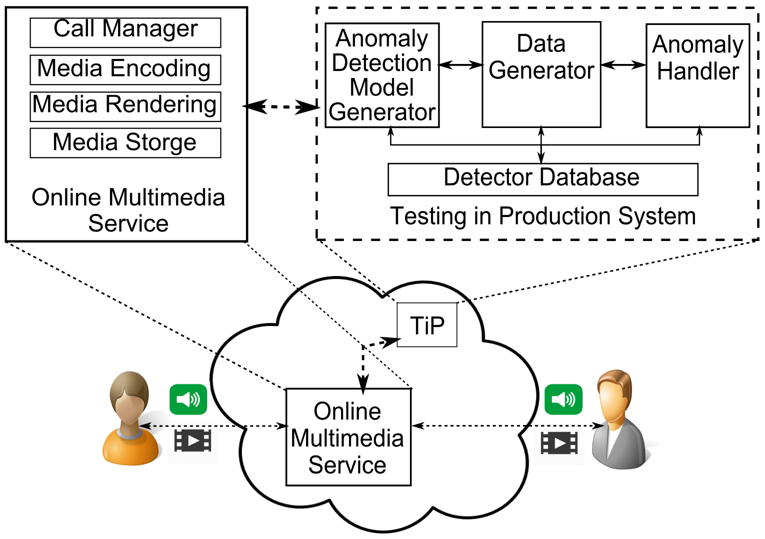
Call Manager
Media Encoding
Media Rendering
Media Storge
Online Multimedia Service
Anomaly Detection Model Generator
Data Generator
Anomaly Handler
Detector Database
Testing in Production System
TiP
Online Multimedia Service

Figure 5.1: High-level architecture of multimedia communication services.

As shown in Figure 5.1, DAA is a Testing in Production (TiP) service to improve the reliability of multimedia services. If DAA or TiP goes down, the service continues to function properly. TiP is common practice in modern online services, especially in interactive multimedia services [14], as without it the service is flying blind. DAA consists of three main components: **Data Generator**, **Anomaly Detection Model Generator**, and **Anomaly Handler**. The Data Generator creates synthetic transactions that replay actual customer transactions that had anomalies in their inputs from the past 30 days, rolling window, of the service deployment. The anomalous values, i.e., the outlier values of service inputs like number of users and number of video sessions, are the values that are found from the trailing 30 day window as encountered by the service. These values are used in the synthetic transactions made by the data generator. By running these synthetic transactions with anomalous inputs and loads, the Data Generator collects the current reaction of the service under the current working conditions like CPU utilization, memory consumption, and number of processes, as well as the service outputs like number of active multimedia connections and their quality measured in MOS.

The Anomaly Detection Model Generator uses the measurements from the Data Generator to establish correlations between the anomalous service inputs, service working conditions, and system states with the service output. The service output is represented by a metric of interest like MOS for shared media quality, service response time, or communication lag time. The correlations are built using current data from the system. We leverage concepts from the *Association Rule Machine Learning* algorithm that are able to determine the level of independence of each input and working condition and their individual impact on the output of the service, as well as the associative impact of multiple inputs and working conditions. The *Anomaly Handler* uses the model and correlations generated by the Model

Generator to identify the groups of anomalous inputs and working conditions that cause SLA violations, classify the anomalies based on their impact, and raise alerts on Impactful anomalies that result in SLA violations.

The following sub-sections detail the functionality of the Data Generator, Anomaly Detection Model Generator, and Anomaly Handler.

5.3.2 Data Generator

The Data Generator is comprised of two components. The *Synthetic Transaction Provider (STP)* and the service *Item and Feature Evaluator (IFE).* The STP makes Testing in Production (TiP) API calls to each component of the service, and uses the anomalous service input values that were found in the service in the trailing 30 days. The service is assumed to have basic anomaly detection, as described in [17], and logs these anomalous values in a database that is accessible by the STP. The STP uses these anomalous service input values in the synthetic loads it generates; for example the number of users, number of sessions created every minute, and the number of videos shared. The STP generates service calls to each component, like the encoder, decoder, dejitter, renderer, and storage components, and passes them the anomalous test loads, and collects their outputs into the Detector Database, shown in Figure 5.1. The service output represents the current service reaction to the anomalies in the service inputs. The data in the Detector Database is real time data, that is generated currently from the system; the lifecycle of this data is in the order of minutes to help monitor the service components. All components of DAA have access to the Detector Database.

In addition to the component synthetic transactions, the STP makes *scenario calls* that represent a true customer e2e transaction. For example, the STP emulates a video call of certain length between two test nodes representing two customers, and shares an actual video between them. Such a scenario call exercises the multimedia service components in a way that mimics real user behavior, using anomalous inputs. The STP collects the results and outputs of these tests into the Detector Database. Test loads to mimic user behavior are generally acceptable to replicate the characteristics and metadata of user transactions. However, if the service at hand requires an exact replica of the user behavior and loads, then parts of previous production logs representing that behavior can be replayed as described in [37, 26]. The STP makes *operating system calls* to collect system information such as CPU utilization, memory consumption, and process counts. The STP runs the component and scenario tests with progressively larger loads to generate actual SLA violations in the tested service and its components. The combination of regular and anomalous inputs, working conditions, and service outputs under low, medium, and high loads are collected into the Detector Database.

Before we describe the Item and Feature Evaluator (IFE) component, we present a few concepts in real-time Dimensional Modeling [35] and Associative Rule based machine learning techniques [57] as they relate to multimedia services:

- **Feature:** is an individual measurable property of a phenomenon or transaction. For example, MOS representing the quality of a multimedia session, is a Feature. Users of DAA provide the Features of interest, like MOS, as a configuration value of DAA.
- **Item:** is a property representing a transaction attribute; for example number of users, number of processes, CPU utilization, and memory utilization are all transaction *Items*. Items map to dimensions in Dimensional Modeling. *Itemset* is the group of Items in a transaction, and maps to Groups in Dimensional Modeling. This allows the study of the root-cause analysis between the Items/dimensions and Features/measures, i.e., which Item values caused which Feature values, as enabled by Dimensional Modeling [35].
- **Transaction:** is the activity of interest, like a multimedia session.
- **Support:** is the frequency an Item is seen in the multimedia session under study. Example: if an Item is shown 8 times in 10 transactions, the Support of that Item is 80%.
- **Confidence:** is the frequency that a deduction is found to be true in the multimedia sessions of study. Example: if a deduction like MOS is below 4 every time CPU utilization is above 77% is found to be true in 900 out of 10,000 transactions, the Confidence in such a deduction is 90%. This data is found in the Detector Database, and Confidence computation is a matter of counting the transactions and their content. Users of DAA provide their required Confidence level as a configuration value of DAA.
- **Lift:** is the ratio between the Support of two Items in the set to the Support of both Items in the set if they were independent. Users of DAA provide their required Lift as a configuration value of DAA. The Lift for Item X and Item Y is given by: *(Support of Union of X and Y) / (Support(X) * Support(Y))*
- **Conviction:** is the ratio of the expected frequency that Item X (e.g., CPU Utilization) happens without Item Y (e.g., Memory Utilization) and causes the multimedia session Feature of interest to happen (e.g., MOS = 5). It is given by: *conv(X, Y) = ((1 - Support(Y)) / (1 - Confidence(X, Y)))*

After the data is generated by the STP, the IFE finds the Feature(s) in each multimedia session in the database. The IFE then finds the Items; these are the remaining multimedia session attributes excluding the Feature(s). It then computes the Support, Confidence, and Lift of each Item in the Itemset of each multimedia session. If the computed values meet the

Procedure 3 Data Generation Algorithm

```
ANOMALY DETECTION DATA GENERATION
 1: function GenerateAnomalyDetectionData
 2:     while (Confidence < ConfidenceConfiguration and Lift < LiftConfiguration do
 3:         STPGenerateData();
 4:         for each Multimedia Session; do
 5:             for each Items do
 6:                 Compute Support, Confidence, and Lift;
 7:             end for
 8:         end for
 9:     end while
10:     Collect Items, Features, Support, Confidence, Lift;
11:     Populate CollectorDatabase with Anomaly Detection Data;
12: end function
SYNTHETIC TEST PROVIDER
 1: function STPGenerateData
 2:     for each Low Service Loads to SLA-Violation-Causing Loads do
 3:         Run component level tests;
 4:         Run scenario level tests;
 5:         Run system level tests;
 6:         Capture test inputs, system states, component outputs and Store in DetectorDatabase;
 7:     end for
 8: end function
```

configuration values for Confidence and Lift, data generation is complete. If the computed Confidence and Lift for any Item are lower than the configured values, the IFE requests more tests from the STP to provide more correlation data that can achieve the required Confidence and Lift. The new STP tests use service input load values that were not used in the previous tests, i.e., it extends the range of used inputs and their load values to ensure that the newly generated data can generate new correlations. The previous inputs and loads are stored in the Detector Database, and are updated after each test. The DAA Data Generation algorithm is summarized in Procedure 1.

5.3.3 Anomaly Detection Model Generator

The multimedia session Features and Items and their correlations as computed by the metrics of Support, Confidence, and Lift are determined by the DAA Data Generator, as described in the previous subsection. The DAA Anomaly Detection Model Generator evaluates each Item in the multimedia session Itemset and their impact on the Features, and computes the Conviction associated with each Item like number of users and their dependent Items like CPU Utilization and Memory Utilization. The Conviction value between two items determines how independent they are from each other. For example, a Conviction between two Items like number of users and CPU utilization of 1.15 means that the correlation between them and the quality of multimedia session is 85% more accurate than the correlation between each of them alone with the quality of service. In other words,

Procedure 4 Model Generation Algorithm

```
ANOMALY DETECTION MODEL GENERATION
 1: function GenerateAnomalyDetectionModel
 2:     for each Feature in DetectorDatabase do
 3:         Group Transactions by Feature Value;
 4:     end for
 5:     for each Feature in DetectorDatabase do
 6:         for each Transaction per Feature Value do
 7:             Group each Transaction Item into Transaction Itemset;
 8:         end for
 9:         for each Item in Transaction Itemset do
10:             Compute Conviction;
11:         end for
12:     end for
13:     Generate Anomaly Detection Model (Table 5.1);
14:     Publish Model to DetectorDatabase;
15:     Publish Conviction of each Item in Transaction Itemset;
16: end function
ANOMALY HANDLER ALGORITHM
 1: function RaiseAlerts
 2:     Define Anomaly-Based Alert Levels (1 to $N$);
 3:     Create a Mapping between Feature measurement and Alert Level;
 4:     for each Real-Transaction Anomalies that cause SLA violations do
 5:         Find the Alert Level Mapping to the Feature measurement
 6:         Raise the appropriate Alert Level
 7:     end for
 8: end function
```

building an anomaly detector that would fire alerts based on values of one of these Items alone without association with the other has an 85% chance of raising a false positive alert. Users of DAA may configure it to consider items independently if the convection between them is above 1.85 for example.

The Model Generator produces a set of tables of associate Items and their ranges that correlate to a given Feature value like media quality MOS = 1, 2, 3, 4, and 5. Table 5.1 contains a sample correlation between the upper bound of number of users and CPU utilization, given a Conviction configuration of 1.85 that results in MOS Feature values of 3, 4, and 5. Table 5.1 in practice is a high cardinality table, and can be Normalized into multiple tables; each representing one Feature, or even one Feature value. Anomalous inputs that cause SLA violations, like MOS 3 or less, are considered impactful and worth raising alerts. Anomalous inputs that result in acceptable measurements of interest, like MOS 4, are considered borderline anomalies that require monitoring but not alerting. Anomalous inputs that do not impact the measurement of interest, like MOS 5, are ignored. The Anomaly Detection Model Generator algorithm is summarized in Procedure 2.

User-CPU Conviction	User Limit	CPU Limit	Video Quality (MOS)
1.85	110	73%	3
1.85	123	61%	4
1.85	128	57%	5

Table 5.1: Anomaly detection model for the multimedia service.

5.3.4 Anomaly Handler

State of the art anomaly detectors are configured to raise alerts when anomalies are encountered [17]. This may result in high number of alerts, with high cost, as we show in the evaluation section later. In DAA, the Anomaly Handler module identifies the anomalies that warrant alerts, and fire alerts when anomalies in the real service inputs and working conditions, i.e., in the Itemset, result in undesired Feature state like MOS 3 or below. DAA users configure the Features to define the measurements of interest and the measurement thresholds at which to fire alerts. Users of DAA configure Lift and Confidence of DAA to define the associative dependence between the service inputs and working conditions, and when to treat them practically independently, and the required level of confidence in the data before firing alerts. By configuring DAA's Features, Lift, and Confidence, users of DAA decide which measurements they want to consider for firing alerts, and the thresholds they consider to be harmful. Anomalies that do not result in harm, like SLA violations, are ignored. This is summarized in the Anomaly Handler algorithm in Procedure 2.

Users of DAA may consider a multi-level alerting system based on the Feature values they are monitoring. For example, if MOS is the Feature of interest, users of DAA may define 3 levels of alerts like Critical for MOS value 1, High Severity for MOS value 2, Medium Severity for MOS value 3. Such sub-classification of the Impactful anomalies is left to the users of DAA to design and implement as they deem fit, which can reduce the cost of handling alerts considerably.

5.3.5 Remarks and Practical Considerations

As explained in the Data Generator section, the computation of the ARL concepts of Confidence, Lift, and Conviction use the value of Support. The computation of Support is quite expensive, and many algorithms like Apriori, Eclat, and FP-growth have been designed to make its computation efficient [18, 17]. State of the art anomaly detection models read massive amounts of data from logs with very low recall, less than 2%, generate connected data graphs, and leverage those algorithms to perform either breadth first search (Apriori), depth first search (Eclat), multi-pass search algorithms (FP-growth) to count the Support for a given feature. These are good optimizations that can make the Support computation tractable. In our proposed approach, we do not need to use any of those search algorithms, because we generate a small amount of data, in the order of mega bytes, as opposed to peta bytes in the logs. The recall in the proposed approach is 100%. The resulting data from the proposed approach is hosted in a dimensional model that lends itself naturally to counting and grouping. This is an important practical advantage of our proposed approach, that drops the machine learning data preparation time from the order of hours or days to a few minutes as shown in the evaluation section. We believe this work is novel because it is the first to combine: (1) synthetic transactions to generate data with high recall in short time, order of seconds, (2) dimensional modeling to identify features and dimensional schema impacting those features, and (3) association rule learning to create an accurate and dynamic anomaly detector, which can be updated in the order of minutes, as opposed to weeks for existing algorithms.

DAA combines machine learning techniques from the Associative Rules Learning (ARL) algorithm and Dimensional Modeling concepts like Features and Groups. The choice of ARL over other algorithms is made to leverage its ability to compute the inter-dependence of parameters contributing to a transaction. Other algorithms are able to find correlations equally well, but lack the ability to find the association between the parameters in the transaction [18, 57]. We combine these techniques with near real-time Dimensional Modeling by defining the transaction output as a monitored Feature, and the Items and Itemsets as Dimensions. This approach and the resulting model are novel and of practical value, when used to correlate the ranges of service inputs and working conditions, with the monitored value of the multimedia service output like video session quality.

Synthetic transactions in TiP do utilize the service resources, and this impacts the service. Service designers account for such an impact due to the importance of TiP [39, 38]. We utilize TiP principles and infrastructures as the platform for DAA. No production code is instrumented to generate the data. Data is generated through TiP synthetic transactions. The Data Generator is executed regularly as part of the TiP system. It collects component, scenario, and system information that are used for generic TiP purposes. The Anomaly Detection Model Generation, on the other hand, is the functionality that takes place on demand, when the anomaly detection accuracy drops below the acceptable value.

5.4 Evaluation

We present the results of running DAA for one month in one of Microsoft's Skype data services, and exercising more than 400 million multimedia sessions.

5.4.1 Implementation and Setup

We implement synthetic transactions for four multimedia components: Call Manager, Media Encoder, Media Renderer, and Media Storage, as shown in Figure 5.1. The considered geo-distributed multimedia communication service processes over 3 million requests per second at peak time. It is deployed in 8 data centers in 3 continents. We use a test cluster of

10 servers in the data center, which gets around 1% of the data center traffic to run our experiments. Each server is a quad-core Intel Xeon server with 12 GB RAM. The STP makes component, scenario, and system calls, and we capture the service components inputs and their outputs. Similarly, we record the outputs of sharing a video scenario. The TiP test cluster we used received an average of 400 million transactions over the four weeks of the experiment, with 3,000 transactions per second at peak. We find the results of the proposed DAA approach from the TiP system, as we capture the test inputs, system states, and the scenario outputs. We find the results of the current anomaly detector of the production service from the system logs, that show the inputs that were considered an anomaly and the resulting output based on that. The production service implements a detector based on Neural Network machine learning. The 1% traffic in the test cluster is split equally between the servers implementing the proposed and current approaches. The traffic split is done by user to ensure continuity of activities received by each system; so 50% of the user base is sent to each system. The results are found for each group and compared.

The STP makes the calls to each of the service components and controls the various aspects of the multimedia request like media type and media size. We measure the quality of the multimedia session using an automated MOS measurement algorithm. Automated MOS measurement algorithms built using actual prior customer assignments that can detect white noise, echo, and other problems are common in test environments that require real-time assessment of media quality [14]. System calls to get system states are implemented in an infinite loop that reads CPU utilization, memory utilization, and number of processes from the performance monitoring APIs of the operating system of each server every 30 seconds. The STP makes simultaneous calls with different user agent information, and controls the load in two ways: (1) number of media sessions made by each client in a given time, and (2) number of simultaneous media sessions representing multiple client calls. The data is collected and stored in the Detector Database with a schema similar to Table 5.1.

DAA can be used as a standalone anomaly detection system. It finds the anomalies and logs them into the Detector Database. However, if the service at hand prefers to find its own anomalies and leverage DAA for analyzing the impact of these anomalies, the service needs to log the anomalies it finds into the Detector Database, so that DAA can use these anomalous values in its synthetic transactions. In the case of our experiments, we used the anomaly values found by the production service, and pulled them into the Detector Database.

5.4.2 Performance Metrics

We study the quality of media sharing during the multimedia session and the number of shared video and audio streams. We compare against the state of the art anomaly detection implemented in the online service using Neural Network machine learning algorithm. The Current anomaly detection used in the service is a Replicator Neural Network detector. It has the classic three phases of: (1) input layer, (2) 5 hidden/internal staircase-like activation layers, and (3) linear output layer. Having 5 internal layers technically classifies it as a deep learning algorithm. The training cycle of the anomaly detector is implemented with backpropagation (i.e. backward pass) for error reconstruction. Here, the error between the actual outputs and the presumed/target outputs is computed, and back-propagated to the hidden layers to update the weights matrix of the neural network neurons. As expected, the training process is lengthy and requires high recall in the training data, which is only guaranteed by large volumes of data from previous runs of the service. The Current anomaly detector implements a preprocessing step, before the replicator neural network, which utilizes a Holt-Winters algorithm for smoothing the time series data representing the service inputs. This step favors fresh data and caters for the seasonality in the data. The following are the metrics we use to assess the performance of DAA:

- **False Positives (FP):** the number of sessions that are considered to have anomaly in their inputs and working conditions, yet did not result in SLA violations.
- **False Negatives (FN):** the number of sessions that are not considered to have anomaly in their inputs and working conditions, yet resulted in SLA violations.
- **True Positives (TP):** the number of sessions that are considered to have anomaly in their inputs and working conditions, and actually resulted in SLA violations.
- **True Negatives (TN):** the number of sessions that are not considered to have anomaly in their inputs and working conditions, and did not result in SLA violations.
- **Accuracy:** the ratio of (true positives + true negatives) to the sum of (false positives, false negatives, true positives, and true negatives).
- **Recall:** in the context of data retrieval from the source, recall refers to the percentage of data that is usable in anomaly detection analysis. In the context of detection analysis, recall is the ratio of (true positives) to the sum of (true positives and false negatives). This the *true positive rate*, or *sensitivity* of the model.
- **Precision:** in the context of anomaly detection analysis, Precision is the ratio of (true positives) to the sum of (true positives and false positives). This is the *Positive Predictive Value* (PPV) of the model.
- **Time to Detect Model Changes:** the time it took the Anomaly Detection Analysis module to detect that the model is no longer accurately representing the current system.
- **Time to Update Anomaly Detection Model:** the time it takes to create a new Dynamic Anomaly Analysis (DAA) model after changes in the system.

Anomaly Detector	FP	FN	TP	TN	Recall	Precision	Accuracy
DAA	10.8	11.5	91.0	93.5	88.8	89.4	89.2
Current	35.2	36.6	50.6	53.4	58.1	58.9	59.2

Table 5.2: Summary of the results over the entire 4-week period.

- **Number of Failures:** the number of sessions that failed to meet the MOS quality SLA due to inaccurate anomaly handling.
- **Media Quality:** the quality of media, audio and video, that is shared between clients; it is measured in MOS.
- **Overhead:** the CPU utilization of the production service with and without the TiP system.

5.4.3 Results

We measure each of the performance metrics described above for the state of the art approach in anomaly detection implemented in the service, we refer to it as Current, and for DAA, and compare the results. First, we summarize the findings of the false positives/negatives, true positives/negatives, recall, precision, and accuracy for the four weeks of the experiment in Table 5.2. The current system and its static way of reacting to anomalies result in huge waste, in the form or false positives and negatives, and the accuracy suffers accordingly. On the other hand, DAA finds the impact of the anomalies in near real-time through synthetic transactions and only alerts if the anomalies result in SLA violations. This reduces false positives and negatives, and enhances the accuracy.

The following figures have only one week of the results, with hourly aggregations of data, to make them clearer. We observed a cyclical pattern daily and weekly, so there are no lost insights by the omission of the remaining three weeks of experiment data from the graphs. We show the detailed graphs for CPU utilization and number of users in the system and analyze the impact of DAA on the performance metrics defined earlier. The proposed DAA approach outperforms the current anomaly detector in all metrics.

Accuracy: In addition to Table 5.2, we detail the accuracy of DAA versus the current detector for users service input and CPU system state, as they are the most impactful on the output of the service. Figures 5.2 and 5.3 summarize the enhancements to CPU and user anomaly detection accuracy. We show the details of accuracy as it provides insights into all the remaining metrics; FP, FN, TP, and TN. Using a static boundary for CPU utilization of 50%, which is what the production service had, results in hundreds of CPU violations per hour that end up being raised as false positive alerts. Using DAA, the accuracy of anomaly detection based on real-time monitoring went up from 59% to 89%. Using DAA, the CPU boundary of safe functionality varied between 40% CPU utilization to 73% before anomalies

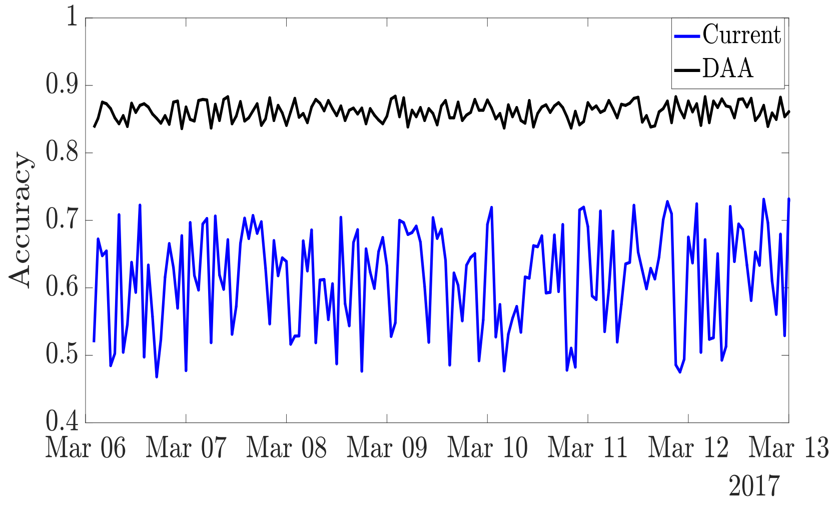

Figure 5.2: Anomaly detection accuracy for CPU.

result in SLA violations. So to assume that we can raise or lower the static boundary, or even use a deterministic cyclical model like sinusoidal to enhance the accuracy, or false positives/negatives, is not true. It needs to be based on data from the current system. Likewise, we see that the accuracy of user anomaly detection using DAA went up from 63% on average to 90%. The memory consumption anomaly detection accuracy went up from 58% to 91%, and the number of sessions anomaly detection accuracy went up from 57% to 89%.

Recall: We compare the findings of data generation time, pre-processing time, and data recall for DAA and the Current detector in Table 5.3. There is significant time saving in the generation of the detector analysis model. Finding and capturing anomalies is an ongoing process throughout the lifecycle of the service. The savings are in the analysis done on the system and its reaction to anomalies. Using current approaches, data about the system needs to be logged for real transactions and then processed and analyzed. Whereas using DAA, we generate small, highly relevant, near real-time data about the current system, not previous deployments of the service. The recall in the original production data, before preparing it for detector model generation was around 2%. This is due to the fact that production logs have a multitude of data describing all aspects of the service like user sign-in/sign-out, authentication, payments, application usage, and other data. It took 22 hours of processing and preparation for the log data to be usable in anomaly detection. DAA has a recall of 100% due to using synthetic transactions that test the required components for the media quality, and the data is usable directly without any pre-processing.

Time to Detect Model Changes: After system changes, like adding new compute resources, it takes DAA at most one minute to detect that the anomaly analysis is no longer accurate for the current system. DAA's Anomaly Detector issues calls to the STP to run

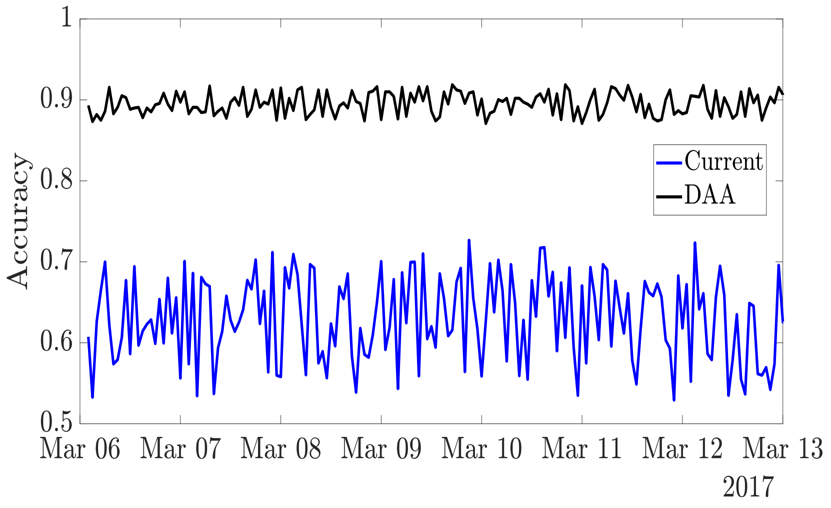

Figure 5.3: Anomaly detection accuracy for user.

Anomaly Detector	Generation Time	Processing Time	Data Recall
DAA	5-7 Mins	2-4 Mins	100%
Current	7 Weeks	22 Hrs	2%

Table 5.3: Data generation and processing times.

more tests in such cases to verify its findings, before it attempts to create new analysis models. We chose three different sets of component, system, and scenario tests to verify. Each run for 5 seconds every 30 seconds. If the model is no longer accurate, we generate a new model.

Time to Update Model: It takes 7-10 seconds to analyze the results of component inputs, system states, and scenario outputs. If the results are close (variance less than 5% in accuracy) from the three tests described earlier, we use the model built from the last set of tests. If the results are not close, DAA assumes the system is still not stable, DAA continues testing the system until it reaches a steady state. On average, it takes 5-7 minutes to update the model. Updating the model usually happens around business day boundaries, and if resources from other data centers are added. During the time of changes, DAA short-circuits itself and lets the production service use its default production anomaly detection service, to avoid introducing TiP-based failures into the production service. When DAA is updated, it requests activation from the production service to perform its anomaly analysis functionality.

Number of Failures: Figure 5.4 shows the number of session failures, quality below MOS 4, caused by inaccurate anomaly reaction with and without DAA. On average, the failures without DAA where around 0.082%. This translates to tens of media sessions failing every hour (63 in 1% of the service traffic); that is thousands of sessions dropping from

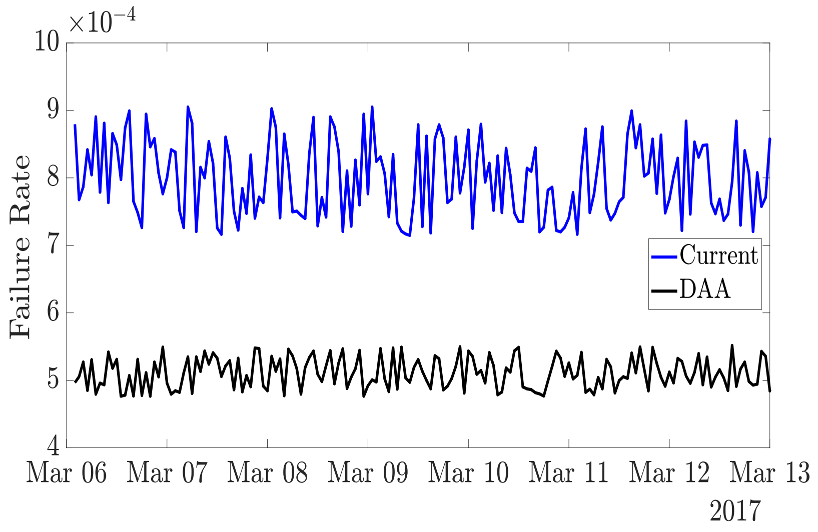

Figure 5.4: Session failures.

MOS 4 or 5, good/excellent quality, to below MOS 4, poor quality, in the data center every hour. Using DAA, SLA violations dropped to around 0.051%. Thus, using DAA results in thousands of customers every hour improving their MOS from poor to good/excellent quality.

Media Quality: Figure 5.5 shows the impact of DAA on media quality enhancement of successful sessions. Around 14% of media sessions have seen an increase from quality of MOS 4, good quality, to MOS 5, excellent quality. DAA reduced the false negative rate from 36.6% to 11.5%. These are inputs that were not supposed to cause SLA violations in the service outputs, but ended up reducing media quality. DAA detected and marked these inputs as anomalies, overriding the Current detector, so the highest safe range of user count per server in a given time window was changed in real-time. This resulted in different routing scheme of new users to other servers. Otherwise, these users would have been added to the wrong server, overloading it, and resulting in compute resource contention and so media quality drop.

Overhead: The overhead of the TiP system is measured by the online multimedia service, continuously. The service measured the hourly average production service CPU utilization with and without the *whole* TiP sytem, which includes DAA. The average service CPU impact caused by the whole TiP system is around 2.8%. The improved multimedia quality, reduced SLA violations, and reduction in false positives and negatives using DAA make the overall investment in the TiP system well justified.

5.5 Summary

In this chapter, we presented a new approach that generates current data about the service in real-time, and uses that data to analyze the impact of anomalies on the service. If the

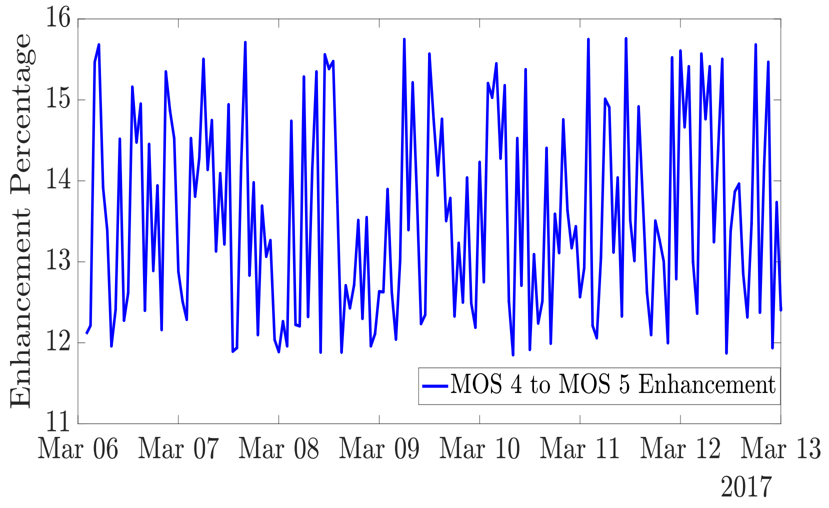

Figure 5.5: Media quality.

inputs and system states do not result in SLA violations, they are not considered anomalies worth alerting on. Through implementation in a production system in one of Microsoft's Skype data services, and running the experiments for 4 weeks, we showed that using the proposed approach reduces the amount of false positives in anomaly detection alerts by an average of 71%, reduces false negatives by 69%, enhances the accuracy of anomaly detection by 31%, and enhances the media sharing quality by 14%.

In contrast, current approaches for anomaly detection, analysis, and handling are static and cannot keep up with the frequent changes that happen during the lifecycle of online services. Our experimental results collected from a large-scale multimedia system show the current approaches result in large waste in the system resources due to the high percentages of false positives and negatives. Current approaches generate many unwarranted alerts that have high maintenance and support cost. This results in poor confidence in the anomaly detection and the alerts they generate.

Chapter 6

Efficient Coordination of Web Services in Large-scale Multimedia Systems

In the previous chapters, we addressed the e2e reliability of online services covering failure prediction, capacity estimation, and anomaly analysis. In this chapter, we address the reliability of online service from a program design perspective. The services and functions of multimedia services like media encoding and dejitter are implemented using a wide range of programming languages. These service are usually run on more than one data center around the world. A common approach for handling the complexity of large-scale multimedia systems is to implement its functionality through *web services*. These web services are called through standard platform-independent protocols such as RESTful APIs.

When a client requests a video conference session, the service constructs and manages a *distributed transactions* that calls multiple web services. The transaction controller attempts to prevent resources from being over-committed. The transaction controller is responsible for the consistency of the transaction. Concurrent transactions cannot interfere with each other, by design. Web services are generally stateless. This makes handling distributed transactions built with web services expensive and complex. The dynamic nature of multimedia services and their constant updates makes the problem harder to manage [3, 4, 5]. All changes and updates made to multimedia web services need to be done without breaking existing client code that is using the multimedia system.

Our work in this chapter addresses the problem of dynamically creating multimedia distributed transactions, and aims at reducing the SLA failures caused by over-committing the service component resources to unnecessary transactions.

6.1 Introduction

The state-of-the-art approaches providing transaction control for distributed transactions in web services are inefficient. They can result in considerable waste of available resources.

This reduces capacity of multimedia session and impacts quality. The limitation in transaction control protocols that causes this inefficiency is the fact that they do not allow participating web services to select which transactions to join. Current protocols force multimedia services to include web services that are not required in some transactions in all distributed transaction. Web services that are not required for the atomicity of a distributed transaction should not be included in one. To make up for the lack of rollback support in web services, multimedia services implement compensating transactions to release the over-committed resources, and to maintain system consistency. Compensating transactions are expensive, hard to implement, and require considerable effort and time in real scenarios [13]. All that result in increased client latency, increased failures, and reduced multimedia session quality.

We propose a simple, practical, and effective *optimization* to current distributed transaction management protocols used in web services. It allows individual web services to *selectively* participate in distributed transactions that they contribute to their successful completions, while fully supporting the dynamic updates of web services, and not requiring any significant changes in the implementation of the web services or the systems using them.

Our implementation in a one Microsoft's Skype data services shows that the proposed optimization substantially reduces the number of web services that are defensively included in distributed transactions, reduce the number of compensating transactions, improves the efficiency of the systems using the web services, enhances the number of multimedia sessions that can be created between users, and enhances the media quality of these media sessions.

The contributions of this work can be summarized as follows: we present an efficient approach to dynamically select which web services to include in distributed transactions in multimedia services. We implement and evaluate the effectiveness of the proposed approach in one of the multimedia communication services used in Microsoft, that handles more than one million sessions per second at peak time. We run the experiments for two weeks in a test cluster that gets more than 200 million multimedia requests. The results show that the proposed approach, on average, increases the throughput of transactions by 36%, reduces failure rate by 35%, the multimedia quality (Mean Opinion Score (MOS)) of the succeeded sessions by 9%, and reduces the overall time required by all transactions by 35%.

6.2 Related Work

Several previous works attempt to address the transaction coordination in distributed multimedia services. For example, Ott et al. [58] address multimedia transaction coordination in distributed services and note the lack of transport-level protocols designed to work independently, as well as inability to share information between multimedia flows without coordinating data transport. They propose an open architecture for sharing network state and transaction information between multimedia flows. Li et al. [59] describe the coordination required for en-route multimedia object caching in transcoding processes for tree networks by requiring service transaction coordination between proxies on a single path or at individual nodes.

Poellabauer et al. [60] argue that real-time and multimedia applications require transaction coordination of event for multimedia delivery mechanism, and note that the observed low Quality of Service in multimedia services is due to lack of effective distributed transaction coordination. Rainer and Timmerer [46] study the impact of geographical distribution of multimedia services and distributed peers, and propose a self-organized distributed transaction synchronization method for multimedia content. They propose a new distributed control scheme that negotiates a reference for the playback time-stamp among participating peers.

Lin et al. [48] study the problems of multimedia distributed transaction latency and their impact on multimedia quality, and introduce compression in graphics streaming. Shatnawi et al. [3] study the use of distributed synthetic transactions to monitor and predict failures in multimedia services. They incorporate a model were all web services are considered part of the transactions, with no ability to dynamically define the atomicity of transactions. Riegen et al. [61] note that scenarios using distributed transactions in online web services are generally controlled by the service client; the service client decides which web services to include in a distributed transaction without any collaboration with the participating web services.

We note that web service distributed transaction management concepts are based on models built for distributed database systems [10, 11]. These models are inefficient for web services. The OASIS projects [13] attempt to define standards for context, coordination, and atomicity between disparate web services. Noting the high cost of transactions in web services, our approach optimizes the selection process of which web services to include in a transaction, builds on top of existing protocols, and adds an optimization communication layer that allows the service client and participating web services to determine, *together*, the web services that need to participate in a given transaction.

6.3 Proposed Optimization

We propose the idea of *selective joining and defecting* from distributed web transactions. This improves the efficiency of distributed transaction protocols by allowing them to dynamically include web services in transactions with minimal overhead and no code changes.

6.3.1 Background

Multimedia services are typically implemented as the collaboration of multiple media web services. The constituent media services may all be part of the same enterprise, or offered through third party online services, such as Amazon Web Services (AWS) [62] for storage.

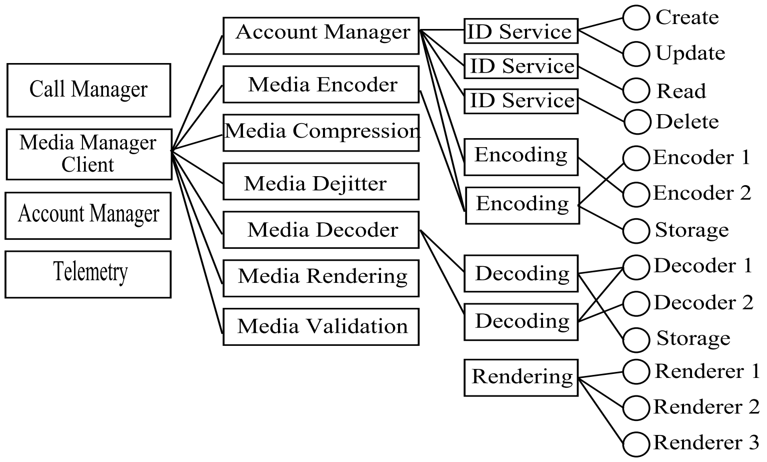

Figure 6.1: Media Manager client and web services.

Figure 6.1 shows a high level diagram of an online multimedia service. Sharing a video between two applications entails getting the user IDs and validating them. The Media Manager Client uses IDAccountManager() that provides create(), read(), update(), delete(), and validate() user accounts. Then the Media Manager Client downloads the video from the source application, performs encoding and size optimization on it, caches the video on the service to enhance the sharing experience and to allow for faster re-attempts in case of delivery interruptions, and finally renders the video for showing at the destination application.

The Media Manager Client uses the following services: MediaEncoding(), MediaDecoding(), MediaStore(), MediaDejitter(), and MediaRender(). The Media Manager Client is the component that controls which web services join transactions. Procedure 1 shows a pseudo code example of how WS Coordination, AtomicTransaction, and BusinessActivity are used in the Media Manager Client to create a transaction context, add web services to it, and conclude the distributed transaction of sharing a video between two applications. To keep the pseudo code simple, we only show how the Media Manager Client creates a transaction context and include web services in it. We show in the following subsections, how the code in Procedure 1 is optimized when we apply the proposed approach.

6.3.2 Overview

From the example in Section 3.1, there are four steps to implement a transaction. (1) Create a transaction context, and register all required web services in it. (2) Execute all web services in the transaction context. (3) Commit the the transaction. (4) Finalize and close the transaction context. The proposed approach works during the first step; we give the web services, e.g., Create, Update, Read, Delete, Encoder1, Encoder2, Storage1, Decoder1, Decoder2, Storage2, Renderer1, Renderer2, and Renderer3 in the Media Manager

Procedure 5 Share a video

```
function EncodeAndShareVideo
    Create WS-CoordinationContext Context
    Create an Activity *ShareVideoActivity*
    Set Coordination Protocol /*e.g. Completion, VolatileTwoPhaseCommit, or DurableTwoPhaseCommit*/
    Set ShareVideoWebServices = Encoding, Storage, and Rendering
    for each Web Service in ShareVideoWebServices do
        Register *Service* in *ShareVideoActivity*
    end for
    Set Compensating Activity for each service in *ShareVideoActivity*
    for each Web Service in Context do
        Call Web Service
        if Web Service Fails then
            Set TransactionFailure = true
            for each Web Service that succeeded do
                Compensate for Web Service
            end for
        end if
    end for
    if TransactionFailure is not true then
        for each Web Service in Context do
            Commit Web Service
            if Web Service Commit Failure then
                Set TransactionFailure = true
                for each Web Service that succeeded do
                    Compensate for Web Service
                end for
            end if
        end for
    end if
    if TransactionFailure is not true then
        Close Transaction Context
    end if
end function
```

example shown in Figure 6.1, *the choice to join* a transaction. The proposed approach uses the existing web service REST APIs in its communications.

Before we explain the proposed approach, we define the software components that participate in it:

- **Service Client**: the service that applications call to perform the multimedia communication, like the Media Manager Client in Figure 6.1. It initiates the transaction and controls its lifecycle and closure; either commit or rollback.
- **Master Service**: the service that has the registration information of all participating web services and the data entities they create, update, and/or delete.
- **Web Service**: the participating web service that represents one atomic functionality provided to the Service Client, like encoding a video.

In the proposed approach, the Service Client calls the Master Service to get information about the web services it is going to call. This includes data entities, like user and video, that the web services write, update, or delete. The Service Client uses this information to make an initial assumption to include the web service in the transaction. The Service Client *may allow* some web services to defect from the transaction if the web service does not impact the transaction data entity, or to join the transaction if it does. This eliminates the inclusion of web services in transactions where they are not needed. This results in more media session capacity and better quality, due to less client latency, faster transaction execution, and higher transaction success rate since fewer web services are included and so less failure points. Less compensating transactions after roll-back in case of transaction failures.

We propose two extra parameters in the methods of the web services that participate in the proposed protocol: (1) a reference to the type of entities that are impacted by the the transaction; e.g., user, video, and audio. (2) An enumeration value that represents the mode of the transaction. The proposed protocol has three modes that control the web service participation in the transaction:

- **All-In**: this is the most common mode in practice in current research and implementation of web service distributed transactions [61, 13]. All participating web services are required to be part of all transactions. If any of them fails, the transaction fails. In the call to the web service, the client passes a reference to the type of entities that will be impacted by the transaction, and a transaction enumeration value of All-In that forces the web service to join the transaction.
- **Defection-Allowed**: the Service Client makes the assumption that the web service is required in the transaction, based on information acquired from the Master Service. The Service Client includes the web service in the transaction in the first step of

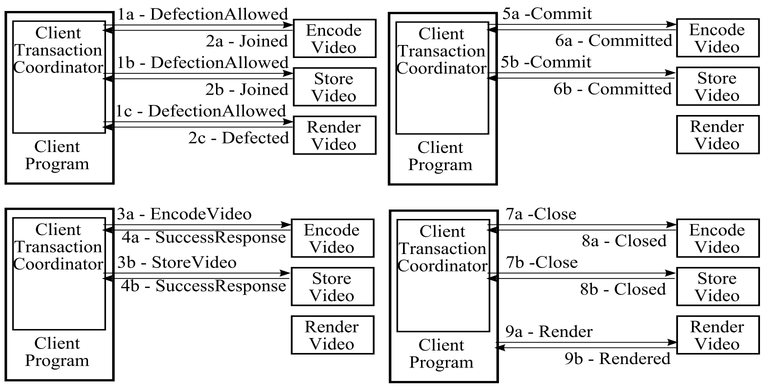

Figure 6.2: Summary of the proposed approach.

Trans-action	Web Service	Entity	Web Service Inclusion	Inclusion Mode
1	Encode-Video	Video	In	Defection-Allowed
1	Store-Video	Video	In	Defection-Allowed
1	Render-Video	Video	Out	Defection-Allowed

Table 6.1: Transaction Participation Table (TPT) example.

creating the transaction context, but allows it to defect if it does not impact the data entity at hand, like user. In the call to the web service, the Service Client passes a reference to the type of entities that will be impacted by the transaction, and a transaction enumeration value of Defection-Allowed. If the web service defects from the transaction, it informs the Service Client through its return value.

- **Join-Allowed**: the Service Client makes the assumption that the web service is not required in the transaction, it does not include it in the transaction context in the initial call, but allows the web service to join the transaction if it impacts the entity at hand. In the call to the web service, the client passes a reference to the type of entities that will be impacted by the transaction, and a transaction enumeration value of Join-Allowed. If the web service joins the transaction, it informs the Service Client through its return value.

Figure 6.2 shows the new sequence of events for the example in Section 3.1 of sharing a video between two applications. Note the impact of our approach on the first and second steps: the Encode(), Store(), and Render() web services are initially included in the transaction context, but are given the chance to defect from the atomic transaction. The Render() web service defects from the transaction, and will be called by the Media Service Client after the Encode() and Store() transaction is successful.

The following sub-sections describe how the Service Client, participating web services, and Master Service implement the proposed protocol.

6.3.3 Service Client Design

The Service Client maintains a *Transaction Participation Table (TPT)* for the transactions it issues. The TPT has the issued transactions, the participating web services, the entities involved, and the web service transaction inclusion mode. If the Service Client assumes a web service is part of a transaction, but allows it to defect, and it defects, then the Service Client updates its TPT to indicate the web service is not part of the transaction. After all web services return from the transaction context creation, Step 1, the TPT is updated to reflect the web services inclusion in the transaction. The Service Client then monitors the success and failure of the web services participating in the transaction, and closes the transaction when done, through commit or roll-back, just as it did without the proposed approach. Note that we did not impact, update or change, the used coordination protocol. If it is Completion, VolatileTwoPhaseCommit, or DurableTwoPhaseCommit, it will proceed as it did; only now with just the *right* set of web services that need to be included in the transaction. Table 6.1, provides an example of an updated TPT that shows the inclusion/exclusion of EncodeVideo(), StoreVideo(), and RenderVideo() in the transaction described in the example in Figure 6.2.

If the Service Client is in doubt about the need to include a web service in a transaction, due to lack of information at the Master Service, the Service Client includes the web service in the transaction using All-In, or Defection-Allowed modes. The Service Client algorithm is summarized in the Distributed Transaction Federated Control (DTFC) algorithm shown in the "DTFC Algorithm - Service Client" procedure.

6.3.4 Participating Web Service Design

Each participating web service registers with the Master Service; the registration process is described in the "Master Service Design" section. The web service checks the available entities in the Master Service Entity Table. If it impacts any existing entity, it adds itself to the entity writers. If the web service impacts entities that are not registered with the master, the web service adds these entities to the master Entity Table, and adds itself as an entity writer.

Each web service adds two parameters to its APIs, the first is a reference to the entities impacted by the Service Client call. The other parameter is the transaction control enumeration described above. It is important to note that by requiring participating web services to report the metadata of the entities they impact, we do not change the statelessness nature of the design and implementation of these web services; i.e. there is no execution state maintained. The participating web services implement the Initialization, JoinTransaction, and DefectTransaction functions as shown in the "DTFC Algorithm - Web Service" procedure.

Procedure 6 DTFC Algorithm - Service Client

SERVICE CLIENT ALGORITHMS START A TRANSACTION

```
 1: function StartTransaction
 2:     Create the Transaction Context
 3:     Create Activity, Specify Protocol, and Register Services
 4:     Get Web Service Table from Master
 5:     Get Entity Table from Master
 6:     Include every web service in the transaction that impacts transaction entities, as found in
    the Master Entity Table
 7:     Create TPT
 8:     Call all Web Services that are included in the transaction
 9:     for each Each Web Service Response do
10:         Compare Returned WebService Inclusion Value with Initial Assumption in TPT
11:         if Inclusion Value is Different then
12:             Update TPT Inclusion Assumptions for Web Service
13:             Update Transaction Service Registration
14:         end if
15:     end for
16:     Wait for all Web Services to Finish or Timeout
17:     if Any Web Service In Transaction Fails or Timeout then
18:         Roll Back through Compensating Transaction
19:     end if
20:     if All Web Services Succeeded then
21:         Commit Transactions
22:     end if
23:     Conclude Transaction and Close Transaction Context
24: end function
```

Procedure 7 DTFC Algorithm - Web Service

```
WEB SERVICE ALGORITHMS INITIALIZE SELF WITH MASTER
 1: function Initialization
 2:     Register Self with Master
 3:     Check Available Entities at Master
 4:     Add Self to Existing Entity Writers That Web Service Impacts
 5:     Add Entities that Web Service Impacts to Master if They Do Not Exist
 6: end function
JOIN TRANSACTION
 1: function JoinTransaction
 2:     Receive Call from Client to Create/Update/Delete Entity
 3:     if Transaction Join Value is not ALLIN then
 4:         if Transaction Join Value is $JOINALLOWED$ then
 5:             if Web Service Impacts $Entities$ then
 6:                 Join Transaction
 7:             end if
 8:         end if
 9:     end if
10: end function
DEFECT TRANSACTION
 1: function DefectTransaction
 2:     Receive Call from Client to Create/Update/Delete Entity
 3:     if Transaction Join Value is not ALLIN then
 4:         if Transaction Join Value is $DEFECTIONALLOWED$ then
 5:             if Web Service Does Not Impact $ENTITY$ then
 6:                 Defect From Transaction
 7:             end if
 8:         end if
 9:     end if
10: end function
```

6.3.5 Master Service Design

The Master Service maintains the following tables:

- **Web Service Table**: has all participating web services names, endpoints, hosting data center, available methods, parameters, authors, readers, writers, and creation date.
- **Entity Table**: has the entity name, ID, description, and web services that update the entity. This table acts as an entity dictionary for the system.

To register, each web service pulls the entity table from the Master Service, updates the table with its entity information, and pushes the updated table back to the Master Service. The Master Service pings the web services regularly to ensure that they are still alive and active. If a web service fails to respond to the Master Service pings after a given threshold, the Master Service removes it from the Web Service Table and from the entity writers in the Entity Table. The Entity Table is an ever-increasing list; there is no need to purge it. Service Clients call the master to get the web service information and the entity lists to allow them to make the right initial assumptions about web services inclusion in their transactions.

6.4 Evaluation

We implemented the proposed approach in the multimedia communication service described in the Background section. It is one of the largest services in the world, and deployed in 8 data centers in 3 continents with more than one million transactions per second at peak. We implement the proposed approach on 6 web services in one data center. Service one, Account Management, manages user information and implements Create(), Read(), Update(), and Delete() user. Service two, Encode Multimedia, manages multimedia information and implements EncodeMultimedia() and VerifyMultimedia(). Service three, Compress Multimedia, manages multimedia size before transmitting it on the wire. Service four, Dejitter Multimedia, manages the de-jittering of audio and video content in live communications. Service five, Decode Multimedia, decodes multimedia content after receiving it at the other end of the communication channel. Service six, Multimedia Rendering, renders the video frames before delivering them in a consumable format to the receiving application.

We run the experiments on a test cluster of 10 servers that get 1% of the data center traffic. Each server is a quad-core intel Xeon server with 12 GB RAM. We deploy the current approach on 5 servers, and deploy the proposed approach on the remaining 5 servers. We implement the Service Client and the Master Service on a separate server. We route the same traffic to both sets of services for two weeks, a total of 200 million multimedia requests, and measure the metrics described below. The average number of distributed transactions generated within these requests is 27%, or around 53 million distributed transactions. The remaining traffic is comprised of requests that do not require setting a transaction.

The metrics we use to evaluate the proposed approach are:

- **Throughput**: the number of transactions handled per second.
- **Execution Time**: the total time used by the system to finish all transactions.
- **Efficiency**: the total computation savings, measured by how many web services are excluded from transactions, as a result of the proposed approach. Note that the excluded web services may still be required as part of the service functionality, but not as part of transactions. So failures in such web services result in re-running them, but not in the rollback of other web services that are needed in transactions.
- **Failure Rate**: the transaction failure rate reduction due to only including web services that are needed in each transaction.
- **Media Quality**: the quality of media (audio, video, and image) that is shared between clients, using MOS. We use a proprietary *automated* MOS algorithm. Automated MOS algorithms enable quality measurements in test environments, where sessions are generated programmatically between test clients.
- **Overhead**: the extra calls incurred by querying participating web services about their impact on the given transaction.

6.4.1 Results

Throughput: 53 million transactions ran over the two weeks of the experiment for a total time of 41 hours and 47 minutes using the proposed approach, and 64 hours and 36 minutes using the current approach. The throughput of the proposed approach is 352 transactions/second, and the current approach is 227 transactions/second. The throughput enhancement is 36%.

Execution Time: The execution time of the proposed approach (41 hours and 47 minutes) is 65% of the current approach (64 hours and 36 minutes). The reduction in execution time is due to the reduced number of web services required per transaction, which we explain in the efficiency section. If any of the web services that is no longer included in a transaction fails, it will not require rolling back of the web services that were included in any transaction. Figure 6.3 shows the daily transaction execution time using the current and the proposed approaches. The enhanced execution time of the proposed approach is 36% on average over the time period of the experiment.

Efficiency: The average number of web services included in transactions in the current approach is 4.6, and in the proposed approach is 2.9. The proposed approach is 37% more efficient in using web services in transactions. These web services would have been

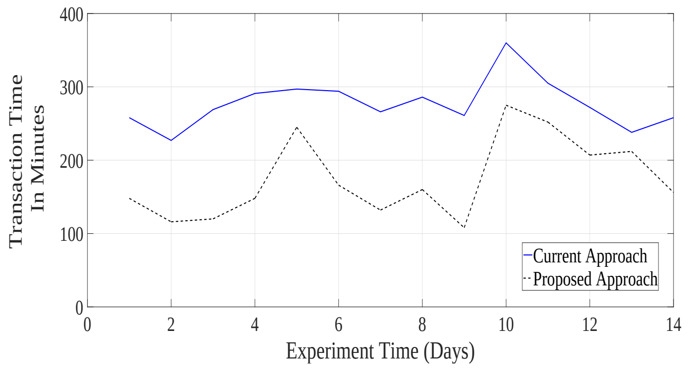

Figure 6.3: Transaction execution time.

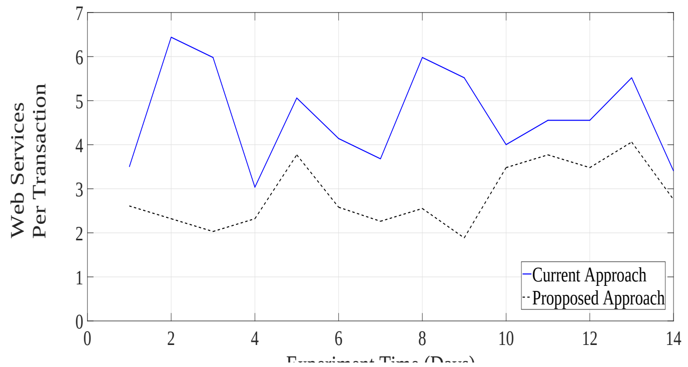

Figure 6.4: Web services per transaction.

otherwise included, unnecessarily and incorrectly, in the transactions. Figure 6.4 shows the daily distribution of web services per transaction, for both approaches.

Failure Rate: From the service logs, the average transaction failure rate due to a web service failure that is not required in the transaction is 17 failures per million transactions. 6 of these, on average, succeeded using the proposed approach since the web service failure did not impact the transaction. The failure rate reduction is 6/17, or 35%, as shown in Figure 6.5. The remaining 11 failures per million were caused by failures in web services that were required in the transactions. We note that the comparison is not perfect, but as close to fair as we possibly could execute it; we pass the same set of transactions to two identical sets of servers implementing the two approaches. The number of web services that fail on the two systems is very close; the difference is less than 2 failures per million transactions.

The enhancements to Throughput and Execution Time of transactions mean more resources are available to handle more media sessions. The reduction of Failure Rate of transactions means the media sessions that would otherwise have failed and required to be reattempted, are now succeeding.

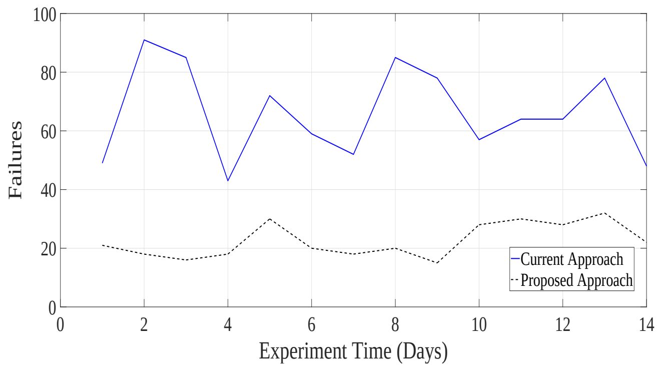

Figure 6.5: Transaction failure rate.

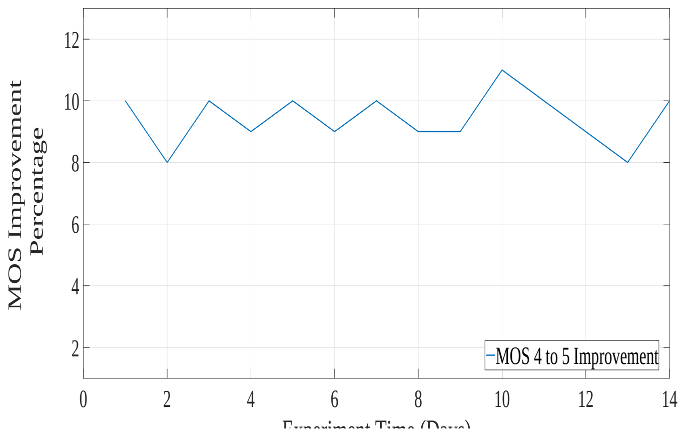

Figure 6.6: MOS increase from 4 to 5 over 14 days.

Media Quality: The succeeded transactions have seen an improvement in their observed media quality (MOS). Figure 6.6 shows that, on average 9% of the sessions that used to meet SLA with good quality (MOS 4), are now meeting SLA with excellent quality (MOS 5). We attribute this enhancement to less number of web services running as one atomic operation, which means more available resources, less load per transaction, and so higher quality.

Overhead: In Defection-Allowed and Join-Allowed modes, the Service Client makes a call to each participating web service to determine its transaction inclusion. The number of these calls and replies are the Service Client overhead. It is found from the average number of web services per transaction without our approach, which is 4.6 calls and their replies. We find the average overhead cost, from the service logs, to be around 0.2ms per transaction. The highest overhead noted was around 0.8ms. The average transaction execution time reduction due to reducing the number of web services in transactions from 4.6 to 2.9 is around 1.6ms. So the average saving of the proposed approach outweighs the overhead of the Service Client by an order of magnitude.

6.5 Summary

In this chapter, we presented a novel approach to allow multimedia web services to selectively join or defect from distributed transactions depending on their impact on the transactions. This reduced the number of multimedia services included in each distributed transaction. In contrast, current approaches to coordinate web service distributed transactions cause client applications to include all possible web services in distributed transactions.This is especially important in multimedia communication services as any waste, loss, or inefficiency in managing resources result in poor multimedia communication quality, which leads to customer dissatisfaction and loss of business.

We evaluated the proposed approach on one of Microsoft's Skype data services, and found that the dynamic coordinator led to enhancing the throughput of multimedia distributed transactions by 36%, and resulted in 9% media quality enhancements from MOS 4 to 5.

Chapter 7

Conclusions and Future Work

In this chapter we summarize the findings of this thesis and discuss potential extensions to this work.

7.1 Conclusions

We presented a comprehensive approach to online service reliability that covers the areas of failure prediction, capacity estimation, dynamic anomaly detection, and web service transaction coordination. We built and verified dynamic approaches to monitoring the health and quality of multimedia communication services. We used *synthetic transactions* to monitor the service by exercising it like real customers, generate current data about it, and then used this *fresh* data to create and maintain the predictive models used in failure prediction, capacity estimation, and anomaly detection of the different components of the multimedia communication service.

Our proposed dynamic approach to creating and maintaining failure predictors for online services in real-time shows superior ability to stay relevant and maintain high accuracy throughout the real-time lifecycle of the online service. Using the proposed real-time dynamic failure prediction (RTDFP) algorithm, we can regenerate an online service failure predictor post system changes in a few minutes with a few megabytes of test generated data.

We utilized the failure predictor effort to test, in real-time, the multimedia service components maximum capacity before failures are observed. Based on that, we were able to build a real-time map of the multimedia component available capacities, and build component selection and routing path actions in real-time to enhance the quality of multimedia sessions. On average, the proposed approach increased the overall media sharing quality by 12%, decreased the percentage of failures by 25%, reduced the CPU usage by 10%, and increased the capacity of the service by 17%.

To overcome the problems of static identifications and definitions of anomalies in multimedia services, we introduced a new approach that generates current data about anomalies in real-time, and used that data to analyze the impact of these anomalies on the service. If the anomalous inputs and system states do not result in SLA violations, they are not considered anomalies worth alerting on. Through implementation in a production system and running experiments for 4 weeks, we showed that using the proposed approach reduces the amount of false positives in anomaly detection alerts by an average of 71%, reduces false negatives by an average of 69%, enhances the accuracy of anomaly detection by an average of 31%, and enhances the media sharing quality by average of 14%.

We addressed the problem of online service coordination between the distributed web services implementing the components of multimedia services. This is especially important in multimedia communication services as any waste, loss, or inefficiency in managing resources result in poor multimedia communication quality, and lead to customer dissatisfaction and loss of business. We presented a novel approach to allow multimedia web services to selectively join or defect from distributed transactions depending on their impact on the transactions. This reduced the number of multimedia services included in each distributed transaction, which led to enhancing the throughput of multimedia distributed transactions by an average of 36%, and resulted in an average of 9% media quality enhancements from MOS 4 to 5.

We demonstrated the use real-time synthetic transactions to monitor online services, generate data representing the current state of the service components, correlate these states and anomalous input with service SLA violations, and enhance the reliability of online multimedia services as a good representative of SaaS service in real-time. This is in contrast to current approaches in literature and industry, which are static and cannot keep up with the changes that happen during the real-time lifecycle of online services. Monitoring the services and waiting for real sessions to fail in order to gain insights into the state of the service is a not an efficient solution.

Static predictive modeling covering failure prediction, anomaly detection, and web service coordination require massive amounts of data to build. Once those models are built, they are suitable for the service systems and working conditions they were built for and based on. If the service systems, provisions, or working conditions change, the predictive models built for them become inaccurate. This requires the process of gathering data, preprocessing it, rebuilding the predictive models, testing them, and deploying them all over again. Static predictive modeling has its strengths and areas of application; they do not require specific knowledge about the system, and are successful in static environments and situations. Online services, however, have dynamic situations that require them to change often.

The generality of the approaches we used comes from the fact that they are system approaches and do not rely on any specific multimedia technology or implementation. They assume generic standard software abstractions for the services and their platforms through RESTful APIs. Any other abstractions are also fine, like RPCs. The implementers do not need to know the internal details of the services at hand. Rather, they only need to know how to call the service through RESTful APIs or standard RPC calls. This makes the approaches usable with any online multimedia service or generic online service. In addition, the approaches we used introduced the concepts of Local Systems and LSTs, Platforms and PSTs, and Scenarios and SSTs. These concepts were implemented using a set of standalone servers as well as in virtualized environments. The concepts are independent of the service models like Platform as a Service (PaaS) and Software as a Service (SaaS). We explained in chapters 3 through 6, that the contributions of this thesis are not limited to a specific technology and/or implementation. Rather, it can be used in the future with new models like SaaS or Software plus Service (S+S). The only thing that will change is how to implement the LST, PST, and SST calls; and that's expected.

In implementing the proposed approaches, we tried to have the setup for the validations and experiments identical as much as possible for all experiments. The goal is to build on previous work, and compare results meaningfully. It was challenging to maintain the same environment setup, and the same usage and traffic load. This was a best effort to the extent possible, and we believe that effort paid off in allowing us to weave a cohesive story for the e2e effort.

7.2 Future Work

The work in this thesis can be extended in multiple directions. In Chapter 3, the failure prediction effort we presented attempts to regenerate a failure predictor using linear predictive models with the shortest possible time to generate, train, and deploy. Advanced predictive models such as ones built using deep learning can be explored to further improve the accuracy of the predictor. An important aspect to consider is the time complexity of such advanced models, as they need to run in real time.

In Chapter 4, we devised an algorithm to measure the maximum capacity that multimedia service components can take before they start to violate their SLAs. We utilized the principles we introduced for predictive failure analysis, and we reduced the time it takes to update the capacity allocations and traffic routing schemes from the order of days to order of minutes. Here too, more advanced predictive models may be utilized to analyze the results of the components' tests in real-time at a faster rate. Having faster analysis can reduce the time to update capacity allocations and traffic routing schemes from the order of minutes to the order of seconds. This can result in enhancing the communication quality of hundreds of calls per hour, and result in that much more customer satisfaction.

In Chapter 5, we analyzed the impact of anomaly detection. Testing for failures around anomalous input, in real-time, should become the norm for all SaaS providers. To help facilitate this, it would be of value to investigate creating real-time synthetic transaction libraries that can be programmed to read anomalous inputs from standard published databases and make standard equidistant time calls to online services, read the output and compare against expected outputs defined in the same databases. Another improvement here, is for SaaS providers of specific types, e.g., credit card unions and travel agencies, to share their learning and identified anomalies through standard libraries and services so that these can be used collectively to enhance the understanding and study of anomalies and their implications. This sharing of anomaly classifications and impact can potentially reduce the effort and time needed by each service to do the same exercise on their own.

In Chapter 6, we introduced a novel approach for web service transaction coordination. We enabled each web service to actively participate in the decision to be included in distributed transactions or not. The approach required slight modification of the service REST APIs to enable the dynamic selection and assignment of a web service based on its impact on the distributed transaction. The effort we did in this area, is probably the first of its kind. We believe this area requires more research and emphasis, due to the waste resulting from the defensive inclusions of all web services in distributed transactions. An area of optimization may be through using a central web service management system that can periodically ping each web service for its role in the defined transactions for the service. This can cover the changes happening to both the web service as well as the transactions. It would be important to investigate the ability to create a universal/global transaction management services that allow different SaaS providers to share such knowledge between them, to enable distributed transactions between enterprises like AWS, Microsoft Azure, and others to optimize their collaboration.

Bibliography

[1] IT Channel. http://www.itchannelplanet.com/.

[2] M. Shatnawi and M. Hefeeda. Efficient coordination of web services in large-scale multimedia systems. In *Proc. of IEEE Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'16)*, Klagenfurt, Austria, May 2016.

[3] M. Shatnawi and M. Hefeeda. Real-time failure prediction in online services. In *Proc. of IEEE INFOCOM'15*, Hong Kong, April 2015.

[4] Z. Li, M. Zhang, Z. Zhu, Y. Chen, A. Greenberg, and Y. Wang. Webprophet: Automating performance prediction for web services. In *Proc. of USENIX Symp. on Networked Systems Design and Implementation (NSDI'10)*, pages 143–158, San Jose, CA, April 2010.

[5] B. Gedik and L. Liu. Peercq: A decentralized and self-configuring peer-to-peer information monitoring system. In *Proc. of the Conference on Distributed Computing Systems (ICDCS'03)*, pages 490–499, Providence, Rhode Island, May 2003.

[6] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. In *Proc. of ACM Computing Surveys, Vol. 42, No. 3, Article 10*, March 2010.

[7] Ben Christensen. *Introducing Hystrix for Resilience Engineering.* http://techblog.netflix.com/2012/11/hystrix.html. and https://github.com/Netflix/Hystrix, 2012.

[8] W. Xu, L. Huang, A. Fox, D. Patterson, and M. Jordan. Detecting large-scale system problems by mining console logs. In *Proc. of ACM Symp. on Operating Systems Principles (SOSP'09)*, pages 117–132, Big Sky, MT, October 2009.

[9] K. Nagaraj, C. Killian, and J. Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *Proc. of USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, pages 353–366, San Jose, CA, April 2012.

[10] S. Bhiri, O. Perrin, and C. Godart. Ensuring required failure atomicity of composite web services. In *Proc. of ACM Conference on World Wide Web (WWW'05)*, pages 138–147, Chiba, Japan, May 2005.

[11] K. Haller, H. Schuldt, and C. Turker. Decentralized coordination of transactional processes in peer to peer environments. In *Proc. of ACM Conference on Information and Knowledge Management (CIKM'05)*, pages 36–43, Bremen, Germany, May 2005.

[12] D. Roman and M. Kifer. Reasoning about the behavior of semantic web services with concurrent transaction logic. In *Proc. of IEEE Conference on Very Large Data Bases (VLDB'07)*, pages 627–638, Vienna, Austria, September 2007.

[13] Oasis web services business process execution language (wsbpel) tc and oasis web services coordination (ws-coordination) and oasis web services atomic transaction (ws-atomictransaction). https://www.oasis-open.org/committees and http://docs.oasis-open.org/ws-tx/wscoor/2006/06 and http://docs.oasis-open.org/ws-tx/wsat/2006/06.

[14] M. Shatnawi and M. Hefeeda. Enhancing the quality of interactive multimedia services by proactive monitoring and failure prediction. In *Proc. of ACM conference on Multimedia (MM'15)*, pages 521–530, Brisbane, Australia, October 2015.

[15] M. Shatnawi and M. Hefeeda. Dynamic anomaly detection in interactive multimedia services. In *Proc. of 26th ACM conference on Multimedia Systems (MMSys'18)*, Amsterdam, Netherlands, June 2018.

[16] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Detecting liveness bugs in systems code. In *Proc. of USENIX Symp. on Networked Systems Design and Implementation (NSDI'07)*, pages 243–256, Cambridge, MA, April 2007.

[17] V. Chandola, A. Bnerjee, and V. Kumar. Anomaly detection: A survey. In *Journal of ACM Computing Surveys (CSUR'09)*, New York, NY, July 2009.

[18] John D. Kelleher, Brian Mac Namee, and Aoife D'Arcy. *Fundamentals of Machine Learning for Predictive Data Analytics.* MIT Press, 2015.

[19] Oracle cloud. https://www.oracle.com/cloud/index.html.

[20] Linux or windows virtual machines in azure. https://azure.microsoft.com/en-us/free/virtual-machines/.

[21] Amazon web services. https://aws.amazon.com/.

[22] Microsoft azure. https://azure.microsoft.com/en-us/.

[23] Gartner report on saas growth. https://www.gartner.com/en/newsroom/press-releases/2017-02-22-gartner-says-worldwide-public-cloud-services-market-to-grow-18-percent-in-2017.

[24] G suite online agreement. https://gsuite.google.com/intl/en-au/terms/2013/1/premier-terms.html/.

[25] Real cost of poor quality. https://devops.com/real-cost-downtime/.

[26] F. Salfner and S. Tschirpke. Error log processing for accurate failure prediction. In *Proc. of USENIX Workshop on Analysis of System Logs (WASL'08)*, San Diego, CA, December 2008.

[27] M. E. Snyder, R. Sundaram, and M. Thakur. Preprocessing dns log data for effective data mining. In *Proc. of IEEE Conference on Communications (ICC'09)*, pages 1366–1370, Dresden, Germany, June 2009.

[28] J. B. Leners, T. Gupta, M. K. Aguilera, and M. Walfish. Improving availability in distributed systems with failure informers. In *Proc. of USENIX conference on Networked Systems Design and Implementation (NSDI'13)*, pages 427–442, Lombard, IL, April 2013.

[29] A. Viswanathan, A. Hussain, J. Mirkovic, S. Schwab, and J. Wroclawaski. A semantic framework for data analysis in networked systems. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, pages 127–140, Boston, MA, March 2011.

[30] Z. Zheng, Z. Lan, B. H. Park, and A. Geist. System log pre-processing to improve failure prediction. In *Proc. of IEEE/IFIP Conference on Dependable Systems and Networks (DSN'09)*, pages 572–577, Estoril, Lisbon, Portugal, June 2009.

[31] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. In *Proc. of ACM Symp. on Operating Systems Principles (SOSP'05)*, pages 105–118, New York, NY, December 2005.

[32] R. R. Sambasivan, A. X. Zheng, M. D. Rosa, E. Krevat, S. Whitman, M. Stroucken, M. Wang, L. Xu, and G. R. Ganger. Diagnosing performance changes by comparing request flows. In *Proc. of USENIX Symposium on Networked Systems Design and Implementation (NSDI'11)*, pages 43–56, Boston, MA, March 2011.

[33] J. Han, M. Kamber, and J. Pei. *Data Mining Concepts and Techniques.* Morgan Kaufmann, 3rd edition, 2011.

[34] M. Kantarczic. *Data Mining Concepts, Models, Methods and Algorithms.* Wiley and IEEE Press, 2nd edition, 2011.

[35] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Definitive Guide to Dimensional Modeling.* Wiley Computer Publishing, 3rd edition, 2013.

[36] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proc. of Symp. on Networked Systems Design and Implementation (NSDI'04)*, pages 309–322, San Fransisco, CA, March 2004.

[37] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat. Pip: Detecting the unexpected in distributed systems. In *Proc. of Symp. on Networked Systems Design and Implementation (NSDI'06)*, pages 115–128, San Jose, CA, May 2006.

[38] L. Riungu-Kalliosaari, O. Taipale, and K. Smolander. *Testing in the Cloud: Exploring the Practice.* IEEE Software Magazine, 2012.

[39] E. Elliot. Testing in production A to Z - tip methodologies, techniques, and examples. In *Proc. of Software Test Professionals (STP'12)*, New Orleans, LA, March 2012.

[40] Testing in production. http://blogs.msdn.com/b/seliot/archive/2011/06/07/testing-in-production-tip-it-really-happens-and-that-s-a-good-thing.aspx.

[41] F. Provost and T. Fawcett. *Data Science for Business: What you need to know about data mining.* O'Reilly Media Inc., 2013.

[42] Social networking statistics. http://www.statisticbrain.com/social-networking-statistics.

[43] I. Trajkovska, J. Rodriguez, and A. Velasco. A novel p2p and cloud computing hybrid architecture for multimedia streaming with QoS cost functions. In *Proc. of Conference on Multimedia (MM'10)*, pages 1227–1230, Firenze, Italy, October 2010.

[44] S. Tasaka, H. Yoshimi, A. Hirashima, and T. Nunome. The effectiveness of a qoe-based video output scheme for audio-video ip transmission. In *Proc. of ACM Conference on Multimedia (MM'08)*, pages 259–268, Vancouver, BC, Canada, October 2008.

[45] Y. Ito, S. Tasaka, and Y. Fukuta. Psychometric analysis of the effect of buffering control on user-level QoS in an interactive audio-visual application. In *Proc. of ACM Multimedia Workshop on Next-generation Residential Broadband Challenges (NRBC'04)*, New York, NY, October 2004.

[46] B. Rainer and C. Timmerer. Self-organized inter-destination multimedia synchronization for adaptive media streaming. In *Proc. of ACM Conference on Multimedia (MM'14)*, pages 327–336, Orlando, FL, November 2014.

[47] Yanwei Liu, Song Ci, Hui Tang, and Yun Ye. Application-adapted mobile 3D video coding and streaming: A survey. *3D Research*, 3(1):1–6, 2012.

[48] L. Lin, X. Liao, G. Tan, H. Jin, X. Yang, W. Zhang, and B. Li. Liverender: A cloud gaming system based on compressed graphics streaming. In *Proc. of ACM Conference on Multimedia (MM'14)*, pages 347 – 356, Orlando, Florida, November 2014.

[49] S. Shi, C. Hsu, K. Nahrstedt, and R. Campbell. Using graphics rendering contexts to enhance the real-time video coding for mobile cloud gaming. In *Proc. of ACM Conference on Multimedia (MM'11)*, pages 103–112, Scottsdale, AZ, November 2011.

[50] Mean opinion score. https://technet.microsoft.com/en-us/library/bb894481(v=office.12).aspx.

[51] Microsoft application and services group. http://www.microsoft.com.

[52] Automated mean opinion score. http://voip.about.com/od/voipbasics/a/MOS.htm.

[53] Lucia D'Acunto, Jorrit van den Berg, Emmanuel Thomas, and Omar Niamut. Using mpeg dash srd for zoomable and navigable video. In *Proc. of ACM Multimedia Systems Conference (MMSys'16)*, page 34. ACM, 2016.

[54] Jean Le Feuvre and Cyril Concolato. Tiled-based adaptive streaming using mpeg-dash. In *Proc. of ACM Multimedia Systems Conference (MMSys'16)*, page 41. ACM, 2016.

[55] Ahmed Hamza and Mohamed Hefeeda. A DASH-based Free-Viewpoint Video Streaming System. In *Proc. of ACM NOSSDAV'14 workshop, in conjunction with ACM Multimedia Systems (MMSys'14) Conference*, pages 55–60, Singapore, March 2014.

[56] Iraj Sodagar. The mpeg-dash standard for multimedia streaming over the internet. *IEEE Multimedia Magazine*, 18(4):62–67, 2011.

[57] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective (Adaptive Computation and Machine Learning series)*. MIT Press, 2012.

[58] D. E. Ott and K. Mayer-Patel. An open architecture for transport-level protocol coordination in distributed multimedia applications. In *Journal of ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM'07)*, New York, NY, August 2007.

[59] K. Li and H. Shen. Coordinated enroute multimedia object caching in transcoding proxies for tree networks. In *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM'05)*, pages 289–314, New York, NY, August 2005.

[60] C. Poellabauer, K. Schwan, and R. West. Coordinated cpu and event scheduling for distributed multimedia applications. In *Proc. of ACM Multimedia Conference (MM'01)*, pages 231–240, New York, NY, 2001.

[61] M. Riegen, M. Husemann, S. Fink, and N. Ritter. Rule-based coordination of distributed web service transactions. In *Proc. of IEEE Transactions on Services Computing (SC'10)*, pages 60–72, 2010.

[62] Transaction library for dynamodb. http://aws.amazon.com/blogs/aws/dynamodb-transaction-library.