Reducing Long Tail Latencies in Geo-Distributed Systems

KIRILL L. BOGDANOV

Licentiate Thesis in Information and Communication Technology
School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden 2016

KTH School of Information and
Communication Technology
SE-164 40 Kista
SWEDEN

TRITA-ICT 2016:32
ISBN 978-91-7729-160-2

Akademisk avhandling som med tillstånd av Kungl Tekniska högskolan framlägges till offentlig granskning för avläggande av teknologie licentiatexamen i Informations- och kommunikationsteknik tisdagen den 29 nov 2016 klockan 13.30 i Sal C, Electrum, Kungl Tekniska högskolan, Kistagången 16, Kista.

Tryck: Universitetsservice US AB

Abstract

Computing services are highly integrated into modern society. Millions of people rely on these services daily for communication, coordination, trading, and accessing to information. To meet high demands, many popular services are implemented and deployed as geo-distributed applications on top of third party virtualized cloud providers. However, the nature of such deployment provides variable performance characteristics. To deliver high quality of service, such systems strive to adapt to ever-changing conditions by monitoring changes in state and making run-time decisions, such as choosing server peering, replica placement, and quorum selection.

In this thesis, we seek to improve the quality of run-time decisions made by geo-distributed systems. We attempt to achieve this through: (1) a better understanding of the underlying deployment conditions, (2) systematic and thorough testing of the decision logic implemented in these systems, and (3) by providing a clear view into the network and system states which allows these services to perform better-informed decisions.

We performed a long-term cross datacenter latency measurement of the Amazon EC2 cloud provider. We used this data to quantify the variability of network conditions and demonstrated its impact on the performance of the systems deployed on top of this cloud provider.

Next, we validate an application's decision logic used in popular storage systems by examining replica selection algorithms. We introduce GeoPerf, a tool that uses symbolic execution and lightweight modeling to perform systematic testing of replica selection algorithms. We applied GeoPerf to test two popular storage systems and we found one bug in each.

Then, using traceroute and one-way delay measurements across EC2, we demonstrated persistent correlation between network paths and network latency. We introduce EdgeVar, a tool that decouples routing and congestion based changes in network latency. By providing this additional information, we improved the quality of latency estimation, as well as increased the stability of network path selection.

Finally, we introduce Tectonic, a tool that tracks an application's requests and responses both at the user and kernel levels. In combination with EdgeVar, it provides a complete view of the delays associated with each processing stage of a request and response. Using Tectonic, we analyzed the impact of sharing CPUs in a virtualized environment and can infer the hypervisor's scheduling policies. We argue for the importance of knowing these policies and propose to use them in applications' decision making process.

Keywords: Cloud Computing, Geo-Distributed Systems, Replica Selection Algorithms.

Sammanfattning

Databehandlingstjänster är en välintegrerad del av det moderna samhället. Miljontals människor förlitar sig dagligen på dessa tjänster för kommunikation, samordning, handel, och åtkomst till information. För att möta höga krav implementeras och placeras många populära tjänster som geo-fördelning applikationer ovanpå tredje parters virtuella molntjänster. Det ligger emellertid i sakens natur att sådana utplaceringar resulterar i varierande prestanda. För att leverera höga servicekvalitetskrav behöver sådana system sträva efter att ständigt anpassa sig efter ändrade förutsättningar genom att övervaka tillståndsändringar och ta realtidsbeslut, som till exempel val av server peering, replika placering, och val av kvorum.

Den här avhandlingen avser att förbättra kvaliteten på realtidsbeslut tagna av geo-fördelning system. Detta kan uppnås genom: (1) en bättre förståelse av underliggande utplaceringsvillkor, (2) systematisk och noggrann testning av beslutslogik redan implementerad i dessa system, och (3) en tydlig inblick i nätverket och systemtillstånd som tillåter dessa tjänster att utföra mer informerade beslut.

Vi utförde en långsiktig korsa datacenter latensmätning av Amazons EC2 molntjänst. Mätdata användes sedan till att kvantifiera variationen av nätverkstillstånd och demonstrera dess inverkan på prestanda för system placerade ovanpå denna molntjänst.

Därnäst validerades en applikations beslutslogik vanlig i populära lagringssystem genom att undersöka replika valalgoritmen. GeoPerf, ett verktyg som tillämpar symbolisk exekvering och lättviktsmodellering för systematisk testning av replika valalgoritmen, användes för att testa två populära lagringssystem och vi hittade en bugg i båda.

Genom traceroute och envägslatensmätningar över EC2 demonstrerar vi ihängande korrelation mellan nätverksvägar och nätverkslatens. Vi introducerar också EdgeVar, ett verktyg som frikopplar dirigering och trängsel baserat på förändringar i nätverkslatens. Genom att tillhandahålla denna ytterligare information förbättrade vi kvaliteten på latensuppskattningen och stabiliteten på nätverkets val av väg.

Slutligen introducerade vi Tectonic, ett verktyg som följer en applikations begäran och gensvar på både användare-läge och kernel-läge. Tillsammans med EdgeVar förses en komplett bild av fördröjningar associerade med varje beräkningssteg av begäran och gensvar. Med Tectonic kunde vi analysera inverkan av att dela CPUer i en virtuell miljö och kan avslöja hypervisor schemaläggningsprinciper. Vi argumenterar för betydelsen av att känna till dessa principer och föreslå användningen av de i beslutsprocessen.

Nyckelord: Datormoln, Geo-Fördelning System, Replica Val Algoritmer.

Acknowledgements

I would like to express my sincere gratitude to my advisors Prof. Dejan Kostić and Prof. Gerald Q. Maguire Jr. I am in debt for their time, effort, and endless patience in guiding me on my research journey. They offered me an invaluable opportunity to learn and improve by working in their group.

I want to thank Miguel Peón-Quirós for introducing me to research and showing me what it means to be a Ph.D. student. I am grateful to Peter Perešíni for his advices, rigorous code reviews and development practices that saved me on many occasions. I would like to express my special thanks to my Ph.D. colleague Georgios Katsikas, for answering my countless questions and tolerating me over these years! I thank Douglas Terry, for being our shepherd for GeoPerf's conference publication [1]. Finally, I want to say thanks to all the people who helped me along the way: Voravit Tanyingyong, Tatjana Apanasevic, Divya Gupta, Robert Allen, Jason Coleman and all the people in the NSLAB!

Kirill L. Bogdanov,
Stockholm, October 31, 2016

The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement 259110.

Contents

Acronyms

CMA	Cumulative Moving Average. 33, 35, 40, 43, 45, 47, 55
ECN	Explicit Congestion Notification. 131
EDR	Exponentially Decaying Reservoir. 33, 39, 40, 43, 47, 50, 53, 78
EWMA	Exponentially Weighted Moving Average. 33, 35, 39, 40, 43, 47, 55, 56, 76, 78, 80, 82
FR	Front Runner. 60, 63, 64, 74–76, 78, 80, 81, 83, 84, 136, 137
I/O	Input/Output. 10, 13, 107, 109, 112, 116
ICMP	Internet Control Message Protocol. 22, 65, 89, 94–96
IMP	Interval Measurement Probe. 108, 112, 114, 116, 118
ISI	Inter-Sample Interval. 119–121, 123, 124
NIC	Network Interface. 14, 106, 121, 141–143, 146, 148
NPP	Network Path Pair. 69, 70
NTP	Network Time Protocol. 64, 93, 96
NUMA	Non-Uniform Memory Access. 108
OS	Operating System. 13, 141
OWD	One Way Delay. 59, 60, 70, 132, 137
PC	Path Constraint. 18–20, 38, 40–42
PCA	Principal Component Analysis. 138
PCPU	Physical CPU. 106–110, 114, 116, 119

PF	Physical Function. 14
PI	Preemption Interval. 109, 114–116, 120, 121, 123–125
QoS	Quality of Service. 1, 2, 7, 10, 12, 18, 137
RTT	Round Trip Time. 16, 17, 22, 33, 34, 36, 39, 43, 61, 65–69, 76, 78, 128, 131, 133
SLA	Service Level Agreement. 10
SLO	Service Level Objective. 10
SR-IOV	Single Root I/O Virtualization. 14
TTL	Time To Live. 65, 83, 84
VCPU	Virtual CPU. 106–110, 112, 114, 116, 119, 120, 131
VF	Virtual Function. 14
VM	Virtual Machine. 13, 14, 106, 120, 121, 124
VMM	Virtual Machine Monitor. 13
WAN	Wide Area Networks. 2, 5, 7, 31, 37, 97, 138

List of Figures

List of Tables

Chapter 1

Introduction

From the groundlaying research into packet switching and creation of the first experimental networks in 1960s, the Internet as we know it today, came a long way to become a global communication medium. The Internet facilitates communication among a vast set of computing services that play an irreducible role in the modern society. From accessing daily e-mails, reading news or booking flight tickets, these services provide people with the way of communication, coordination, trading, and accessing to information. These services operate on the unprecedented scale, allowing seemingly instantaneous exchange of information between remotely connected locations on this planet, while providing services that were not attainable before.

For example, Jammr[2] provides a means for musicians scattered around the world to play together in real-time, overcoming the physical distances and network latency. Another example is the emerging area of telepresence. In particular, remote surgery[3] opens a possibility for doctors to perform surgeries on patients away over thousands of kilometers. Many of these services and technologies are latency critical and require extensive resilience to external factors to maintain their Quality of Service (QoS) throughout the session.

The ability of these services to provide reliability and performance is the key property that determines the success of an individual service (its popularity and the revenue of a company) and the general spectrum of technologies that is available to us as a society at present (e.g., access to remote surgery).

To meet high demands and expectations, many of today's services are implemented as geo-distributed systems and deployed on top of third-party cloud providers. Cloud providers maintain datacenters in multiple geographic locations to provide a common, virtualized platform that allows flexible, worldwide deployment of applications. The services can thus be geo-replicated across datacenters to provide services to clients in different regions, share workload, and to prevent critical data loss in the case of a failure or disaster occurring at any

given datacenter.

For example, Gmail, the Picasa image sharing website, and Google Apps (e.g., Google Calendar, Google Docs, etc.) are based on a strongly consistent database called Google Spanner[4]. In Spanner, users' updates are replicated via the Paxos[5] consensus protocol across machines running in multiple datacenters, spread over different geographic regions. The global throughput of any synchronous database is dependent on the speed at which distributed nodes can come to an agreement to execute an action (e.g., end-user request). Therefore, it is extremely important to quickly and efficiently propagate update messages from the source node (the entry point in the service where the request is received) to a carefully selected quorum of nodes.

However, the nature of geo-distributed deployments on top of virtualized environments of cloud providers exhibits unstable performance. For example, diurnal user access patterns create uneven loads on servers located in different regions, network conditions across Wide Area Networks (WAN) constantly change due to competing traffic of other services, and failures in one region can affect the whole service.

To compensate for these changes, geo-distributed systems try to adapt to changing conditions by monitoring changes in network and system states. This is often done by measuring network latency and load on remote replicas (via measuring the CPU utilization or the number of outstanding requests).

The combination of variable factors poses difficulty for a distributed system to determine the best set of actions be it replica selection, server peering, or quorum selection. Moreover, a suboptimal decision will degrade the service's overall performance and potentially violate the desired QoS. Failure to cope with the high dynamicity of changing network conditions, changes in quorum, and varying loads can introduce disruptions and failures in many on-line services that users rely upon.

In this work, we attempt to improve the quality of geo-distributed services by improving the quality of run-time decisions made by these systems. Our approach is based on extensive network and system measurements performed in a cloud provider. We address both the core logic involved in making run-time decision and the perception of the network and system states that acts as a primary input into the core logic.

In Section 1.1 we summarize our research objectives and highlight the main targets of this work. Next, in Section 1.2 we describe our research methodologies and techniques. In Section 1.3 we summarize the contributions of this work. In Section 1.4 we discuss sustainability and ethical aspects of this research. Finally, in Section 1.5 we give a bird's-eye view of the rest of this thesis.

1.1 Research Objectives

In this thesis, we seek to improve geo-distributed systems that facilitate popular services and critical cloud infrastructure. Our main research objectives are tailored towards improving the performance and reducing tail response latency of geo-distributed systems deployed on third party public cloud providers' infrastructure. In particular, we narrowed the scope of the problem to improve systems' adaptability towards dynamically changing conditions in the Internet and public clouds. Therefore, our primary objective can be summarized as:

- ***Objective 1:*** *Improve the quality of run-time decisions made by geo-distributed systems.*

Our secondary research objective is designed to measure and understand the underlying deployment conditions for geo-distributed cloud providers' infrastructure. This information will tell us the degree of resilience and adaptability that is required for a geo-distributed system to provide high performance under changing conditions.

- ***Objective 2:*** *Measure and analyze the degree of performance variability experienced by geo-distributed systems, deployed on a modern public cloud provider.*

Throughout this work, measurements play a critical role by providing a view of the geo-distributed cloud infrastructure in terms of network latency, routing, and virtualization aspects of the cloud's environment. All the data obtained in this work is to be publicly released to support other researchers working in this area.

1.2 Research Methodology

This thesis project uses the empirical research approach. Throughout this work, we performed a set of measurements and observations that describe the state and properties of real-world systems. The collected data was analyzed to improve our understanding of geo-distributed systems and cloud platforms. The knowledge obtained from our measurements was used to design solutions for each given problem that we addressed in the problem's domain. As needed, additional measurements were performed during the development cycle. After each implementation phase, our solutions were deployed and evaluated either in a real-world setting or (if not possible) within our laboratory environment (using trace replay and simulations). During the evaluation phase, we obtained empirical evidence and measured the effectiveness of each proposed approach.

Our approach towards improving the quality of run-time decisions made by geo-distributed systems is based on the following steps:

1. Measure and evaluate the deployment conditions across a third-party cloud provider. We performed a set of quantitative measurements across Amazon EC2 to understand the degree of variability in network conditions. These observations provided us with a view of network latency across geo-distributed datacenters of this popular cloud provider. The data was subsequently used to test our hypothesis and perform trace replay evaluations. The analysis of this data dramatically improved our understanding of the underlying network conditions that geo-distributed systems face today.

2. Derive techniques for systematic verification of the core logic responsible for the decision making process. Continuously changing conditions within geo-distributed deployments pose a challenging problem, thus making run-time decision inherently difficult. We attempt to address this problem by providing novel techniques of systematic testing and comparing core logic algorithms employed in the decision making process.

3. Validate input parameters that are used by the core logic in making run-time decisions. Systems performing run-time decisions rely on having an accurate view of the network and system states. In this step, we derived techniques to improve the quality of data representing these states. We exposed implicit information about network and system states and make it explicit, thus allowing the geo-distributed system to make informed decisions.

4. Address implications of virtualized deployments. The virtualized environment offered by cloud providers incurs performance issues due to hardware sharing and multiplexing. This results in performance variability of systems deployed in such an environment. In this step, we quantified the impact of CPU sharing on Amazon EC2 and inferred XEN hypervisor's scheduling policies in this setting. We make this information available at run-time to systems and network protocols that operate in this environment. The aim is that applications can make better run-time decisions by exploiting this information.

1.3 Thesis Contributions

As the foundation for this work I performed a set of **network measurements** across Amazon EC2 cloud provider and collected extensive real-world traces. These measurements resulted in the following contributions:

NM1 To date I have performed long-term (14 months) network latency measurements across all ten currently available datacenters of Amazon EC2. This dataset

illustrates temporal changes in network latency across WAN links used by this popular cloud provider. Using this dataset I demonstrated that any static configuration of replicas will perform suboptimally over time. The first part of this dataset has been publicly released; the remaining dataset is planned to be released by the end of this work*.

NM2 Using trace route and one-way delay measurements across ten geo-distributed datacenters of Amazon EC2, I demonstrated a correlation between unique network paths and network latency. I showed that packets traveling a previously observed network path incur the same network delay as previously observed for the given network path (excluding any additional delay due to network queuing and congestion).

NM3 Using my measurements, I demonstrated that the number of network paths between a pair of geo-distributed datacenters of Amazon EC2 is finite and relatively small. The combinations of possible forward and backward paths that can be taken by packets produce a small number of latency classes (often less than 5 per network path). Moreover, the presence of a particular IP address along the network path can indicate a *persistent* change in latency.

We designed and developed **GeoPerf**, a tool for systematic testing of replica selection algorithms. GeoPerf applies symbolic execution techniques to applications' source code to find potential software faults (bugs).

GP1 GeoPerf uses a novel approach of combining symbolic execution and lightweight modeling to systematically test replica selection algorithms. Using heuristics and domain specific knowledge (obtained from network latency measurements), GeoPerf mitigates state space explosion, a common difficulty associated with symbolic execution, making systematic testing of replica selection algorithms possible.

GP2 GeoPerf addresses a challenging problem of detecting bugs in replica selection algorithms by providing a performance reference point in a form of the *ground truth*. By detecting performance deviations from the *ground truth* implementation, GeoPerf exposes performance anomalies in replica selection algorithms, making it easier to detect the presence of potential bugs.

GP3 We applied GeoPerf to test replica selection algorithms currently used in two popular storage systems: Cassandra[6] and MongoDB[7]. We found one bug in each. Both of these issues have been fixed by their developers in

*The steps to obtain the dataset are available at KTH DiVA http://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A844010

subsequent releases. Using our long-term traces I evaluated the potential impact of the bugs that were found. In the case of Cassandra, the median wasted time for 5% of all requests is above 50 ms.

We designed and developed **EdgeVar**, a network path classification and latency estimation tool. EdgeVar combines network path classification along with active traceroute and network latency measurements to decompose a noisy latency signal into two components: *base latency* associated with routing and *residual latency* variance associated with network congestion. To the best of our knowledge, EdgeVar is the first latency estimation technique that takes into account the underlying network architecture.

EV1 We designed and developed Front Runner, a domain-specific implementation of the step detection algorithm, to identify changes in latency classes based on the latency stream (i.e., the sequence of measurements of latency values) alone. Leveraging the notion of network paths and the minimum propagation latency achievable on each path, on average, Front Runner requires less than 20 latency samples to identify a change in latency classes.

EV2 By exploiting knowledge of the network paths and the associated latency classes, EdgeVar removes the effects of routing changes from the latency stream, allowing clear identification of network congestion.

EV3 By combining knowledge of latency classes and network variance, EdgeVar improves the quality of latency estimation, as well as the stability of network path selection. Using trace replay, I demonstrated that during a period of network congestion, EdgeVar reduces the number of network path flaps, performed by the application, between 6 to 40 times that of conventional techniques.

We designed and developed **Tectonic**, a tool for request completion time decomposition. Tectonic performs user space and kernel space timestamping of the targeted application's messages. Tectonic extends EdgeVar by providing a view into the service time component of application delay. This is an ongoing work and the aim of this work is to provide the following contributions:

T1 Using Tectonic I decomposed Cassandra's service time into components of request propagation delay through the Linux TCP stack and the user space service time. Using this decomposition I demonstrated the relationship between the load on the VM and the request propagation and service time.

T2 Using techniques associated with real-time Linux kernel testing I developed tools to infer the XEN scheduling policies currently used in Amazon EC2. Next, I demonstrated how these policies can also be inferred by using Tectonic's timestamps.

1.4 Research Sustainability and Ethical Aspects

The successful completion of this research will improve the quality of run-time decisions made by geo-distributed systems. To realize this goal, new tools and techniques have been developed that facilitate geo-distributed deployments and improve the QoS. We anticipate that the impact of these advancements might have the following implications.

Economic Sustainability. Economically, this project will produce the tools and techniques that will facilitate deployment of geo-distributed services while reducing their latency and response times. First, this will provide the necessary technological foundation for new type of services to evolve, and as a result, will create opportunities for new business to emerge. Moreover, validating correctness and improving the adaptability of distributed systems to operate in dynamic environments will lower the complexity entry point for many companies to deploy their services across the globe.

Environmental Sustainability. The contributions of this work towards improving network and system awareness in combination with a better decision making logic are likely to have a positive environmental impact.

First, the network communication overhead will be reduced by lowering the redundancy of WAN communications among multiple replicas of a service. Second, load awareness will improve hardware utilization which will lead to the reduction in the number of replicas. Hence, this will lead to a reduction in hardware requirements on a per-service basis and will translate to a reduced energy footprint.

Societal Sustainability. Societally, this research will reduce latency of key cloud systems and communication services. Decreased latency of responses in these critical services directly translates to increased productivity for a large fraction of the population.

By solving challenging problems associated with geo-distributed deployments, this work facilitates the growth of popular services. As a result, it will drive deployment costs down, which will reduce services' fees, making them more affordable for the general public.

Finally, technological advances can result in the development of new services that will become an important part of peoples' lives in the near future.

Ethical Aspects. Throughout this work, we performed extensive measurements and detailed analysis of the third party cloud provider and its WAN characteristics.

We attempted to be very explicit in stating the setup and tools used in each measurement. To facilitate reproducibility of our results important code fragments are available in the Appendixes of this thesis; the remaining source code and all our measurements are available upon request.

Due to rapidly changing computer technology and dynamic nature of WAN it might be nearly impossible to reproduce the exact experimental results measured across Amazon EC2. Still, the underlying phenomena that we discuss in this work should be present and easily verifiable. For example, a correlation between a unique network path and network latency should exist among all cloud providers (see Chapter 5). Similarly, techniques used to infer XEN's CPU scheduling policies across Amazon EC2, should still work across other cloud providers and different hypervisors (see Chapter 6).

Finally, in this work, we developed techniques, which extract system and network state information that was previously implicit (e.g., hypervisor's CPU scheduling policies). However, at no point during our measurements, we breached Amazon's acceptable use policy[8], customer agreement[9] or collected information about other EC2 clients.

1.5 Thesis Organization

The rest of this thesis is organized as follows. Chapter 2 provides the background information necessary to understanding this thesis. In Chapter 3 we describe our cross-datacenter latency measurements as performed on EC2 and then use them to motivate the need for dynamic adaptation towards changing network conditions. In Chapter 4 we describe GeoPerf, our solution for systematic testing of replica selection algorithms. Next, in Chapter 5 we introduce EdgeVar and demonstrate, using additional traceroute measurements across EC2, how we can correlate latency levels with distinct network paths. In Chapter 6 we introduce Tectonic and demonstrate its use by the example of Cassandra. In Chapter 7 we summarize related work and relate it to this thesis. Finally, we conclude this thesis in Chapter 8 and suggest some future work.

Chapter 2

Background

This chapter presents the concepts and techniques used in the rest of this thesis. We study the deployment of geo-distributed applications on top of third party cloud providers. Thus, we begin by introducing the concept of cloud computing and virtualization in Section 2.1.

Next, in Section 2.2 we describe a common replica selection process. The replica selection process used in the systems described in this thesis utilize dynamic run-time decisions made by a distributed system. Finally, in Section 2.3 we describe symbolic execution - a technique for systematic testing and verification of an application's implementation.

2.1 Cloud Computing

The concept of cloud computing is based on the idea of hardware virtualization and dates to the 1960s when IBM introduced a time-sharing virtual machine operating system for their mainframes. Their time-sharing technique provided multiple users with a time-slice of a physical computer, allowing them to execute their programs nearly simultaneously*. At that time, hardware costs were much higher than maintenance cost. This technique solved two problems: (1) it provided access to computational resources for individuals and institutions who could not afford owning their own computer and (2) it simultaneously improved hardware utilization[10].

The idea of time-sharing was re-born in early 2000 as cloud computing, but its main concepts and purposes were unchanged. Cloud computing provides virtualized abstractions of computing resources, including CPU, memory, storage, and networking. These resources can be logically shared among multiple users in a flexible way, while providing a multitude of benefits and reducing costs. A cloud

*Users' programs were executed serially in a rapidly interleaved fashion.

provider maintains a datacenter (or a set of datacenters) which aggregates a large amount of computational resources. These hardware resources are logically divided in order to deploy and run client specific VMs. Such an arrangement removes the need for individuals or organizations to maintain and provision hardware at their own premises, instead all the hardware purchasing and maintenance is outsourced to a cloud provider. Clients pay only for the resources consumed by their VMs.

This new cloud paradigm has revolutionized the way modern services operate and has produced a broad market of private and public cloud providers. Among the most popular public clouds providers are Amazon Elastic Compute Cloud (EC2)[11] (launched in 2006), Google Cloud[12] (launched in 2009), Microsoft Azure[13] (launched in 2010), and IBM Cloud[14] (launched in 2011).

One of the most important benefits of Cloud Computing is its cost efficiency. Sharing hardware among multiple users and applications improves this hardware's utilization and drives the costs of each of these users down. Most cloud providers use pay-as-you-go contracts where clients are charged hourly based on their resource consumption. The granularity of pricing allows for dynamic scaling; hence, applications and services can scale up or scale out by increasing/decreasing the quality or quantity of their VMs on demand, according to their current or anticipated needs. Modern datacenters can provide almost infinite resources, removing the need for clients to perform hardware provisioning and avoiding the need to do capacity planning and hardware acquisition & installation based upon the anticipated peak demand.

A Service Level Agreement (SLA) is a legal contract between a service provider (in this case a cloud provider) and a user. This document defines the responsibilities and the scope of a service. An important subset of an SLA is the Service Level Objective (SLO) which defines the QoS and the exact metrics that are used to evaluate the provided service. For example, a commonly used criteria is up-time or service availability, this value is computed as the percentage of time that a service was operational over a stated interval of time. Other criteria include minimum Input/Output (I/O) throughput, maximum network latency, and minimum network bandwidth. High standards of SLOs by a cloud provider become a differentiating factor as when these are embodied in an SLA they directly affect applications and services that are deployed in the cloud. Therefore, it is common for a cloud provider to pay penalties if SLO metrics are not met [15, 16].

Cloud computing has become a common platform for many systems that are used in our daily services, such as email and calendars, social networks, financial systems, and many more. These services are regularly used by hundreds of millions of people. At the core of these systems are distributed storage systems, specifically: Gmail and Picaso uses Google Spanner [17], Facebook is based on Cassandra[6], LinkedIn uses Voldemort[18], and Amazon Store is based on DynamoDB [19].

Many of these services are latency critical . Increased request delay is negatively correlated with the user satisfaction and as a result, negatively affects a service's revenue and its overall popularity. For example, Amazon has reported that a latency increase of 100 ms causes a loss of 1% of sales [20].

The term **tail latency** defines a fraction (usually less than 1%) of latency (or request completion times) measurements with the longest delay. For example, a 99^{th} latency percentile corresponds to the smallest latency value from the 1 percent of largest samples. The length of the tail is often determined by the ratio of the tail latency to the median (50^{th} percentile) latency; thus if the ratio is large, then the tail is said to be *long*. The tail latency is an important metric describing the worst delays associated with network or system performance. For a popular service, even 1% of requests corresponds to a significant number of customers. Moreover, in the presence of composite requests (i.e., requests containing multiple sub-requests), even a small percentage of delayed queries can have a substantial impact on the system's overall performance[21].

The process of **replication** creates a copy of the application's data (or a subset of the data) and stores it in another physical machine (known as a replica). This is done for reasons of data survivability, availability, and improving the application's performance.

Having multiple copies of the data allows the application to survive failures and disasters at a different scales; if one copy is destroyed or temporary inaccessible, then a copy from a replica can be used instead. If a replica is destroyed or damaged, it can be restored based upon consensus with the surviving replicas. Having multiple copies of the data within a single datacenter provides data availability in the case of a single machine or rack failure. Having replicas spread across multiple geographic regions guarantees data survivability even in the event of a local natural disaster, complete datacenter failure, or a failure in the network connectivity to and from the datacenters and end users or between datacenters. For example, Amazon EC2 utilizes the hybrid concept of an "availability zone", where each regional datacenter is divided into a number (often a triplet) of isolated sub-datacenters some distance apart, each with independent infrastructure. Replicating across an availability zone improves data survivability within a geographic region, i.e., if one sub-datacenter fails, the second replica in the same availability zone can take its place, thus clients located in the same region can still access the service without connecting to a remote geo-distributed replica.

Data replication facilitates scalability by allowing multiple physical machines to operate on the same data (e.g., enabling the service to serve a larger number of clients' requests). Scalability can be generalized into two categories: vertical (also known as scale up/down) and horizontal (scale out/in). Vertical scalability implies increasing/decreasing the resources (i.e., CPU, memory, network) of a

single machine that is operating on the data. While vertical scaling is the simplest solution, because it does not require any modifications to the application logic; scaling-up is bounded by the physical limitations of the available hardware.

In contrast, horizontal scalability increases the quantity of computers that operate on the data. However, depending on the application and architectural decisions, horizontal scaling requires some form of data synchronization and partitioning to allow multiple computers to share the workload. This partitioning may be as simple as sharding (where each machine operate on its own isolated segment of the data) or may require very complex replication management to control how the multiple computers concurrently handle the data.

Having multiple copies of the data in different geographic locations often improves performance by moving the data closer to the end user, hence lowering the response time for nearby clients, thus improving the QoS of the given service as experienced by these users. Unfortunately, accessing the nearest copy of the data does not always provides the fastest response time, as the response time can be affected by the network conditions and the amount of load put on the nearest replica by other users. These parameters needs to be understood, measured, and taken into consideration when choosing the point of contact to a service. Due to the many contributing factors performing replica selection automatically at run-time is difficult.

2.1.1 Geo-Distributed Systems

Major cloud providers have deployed datacenters in multiple geographic regions. For example, to date Amazon EC2, Microsoft Azure, and Google Cloud have datacenters in 10, 24, and 8 geographic locations (respectively). As noted above, utilizing multiple geographic locations improves QoS by replicating data and moving the data closer to the end user (hence reducing part of the latency). As a result, the ability to achieve low latency of responses to requests becomes a differentiating factor for geo-distributed services deployed upon a cloud provider's infrastructure [21, 19].

Unfortunately, the broad range of cloud providers and their services often makes it difficult for a client to choose the most suitable option. All cloud providers differ in the number of datacenters and their geographic locations, service availability, and cost of usage (typically based on numerous metrics of network and hardware utilization) [22, 23]. Moreover, their characteristics change over time by providing new services, new datacenter locations and changing prices.

A broad range of deployment options among cloud providers imply that some clouds are better at certain aspects of their services (be it the price, performance, or geographical presence). A number of studies [17, 24] propose to deploy services across multiple cloud providers simultaneously in order to capture the benefits of

each cloud (also known as a *mutlicloud* deployment). Despite these benefits, to date, such deployments did not gained in popularity, due to security issues and managing overhead.

2.1.2 Virtualization

Cloud computing generally exploits hardware virtualization. This hardware virtualization is typically implemented via a hypervisor (also known as a Virtual Machine Monitor (VMM)) running in an Operating System (OS) or running on the bare metal (directly on the hardware). This hypervisor facilitates deployment of the Virtual Machines (VMs) that share the underlying hardware. The hypervisor provides resource management and communication between VMs and the underlying physical resources (devices). For example, EC2 uses XEN [25], Microsoft Azure uses Hyper-V[26], and Google Cloud uses KVM[27] as their respective hypervisors.

Figure 2.1 demonstrates the concept of virtualization. The bottom layer represents the physical machine with its CPU, RAM, and I/O resources. Next the optional OS layer, followed by hypervisor. A hypervisor can host a set of VMs (also known as guest VMs) each of which has its own OS and runs a set of applications. These applications communicate with the external world through the guest's VM. It is also possible for a hypervisor to be merged with the OS as has been done in Linux containers [28].

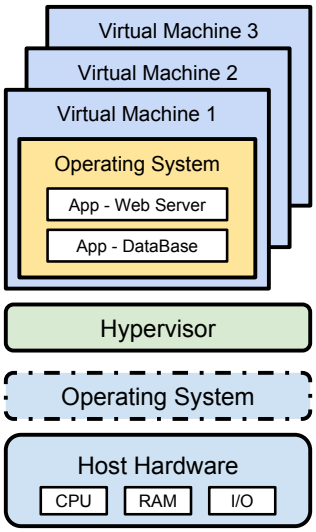


Figure 2.1: Virtualization concept

However, the flexibility offered by virtualization comes at the price of performance. First, the hypervisor layer introduces overhead, while simultaneously consuming some of the system's resources. Second, sharing of a hardware resource can lead to covert channels between VMs sharing this resource[29].

Lots of effort has been made to measure, understand, and mitigate virtualization overheads. Many works have compared different cloud providers[22, 23], virtualization techniques[30, 31, 32] and described their network characteristics [33, 34, 35].

With regard to EC2, these works have generally evaluated older versions of EC2, when it was based on paravirtualization techniques [33]. Today, high-performance EC2 instances use the Single Root I/O Virtualization (SR-IOV) [36] virtualization technique. SR-IOV removes the hypervisor from the process of moving packets from the Network Interface (NIC) to a guest OS and from the guest OS to the NIC. This is possible because SR-IOV provides access to a portion of a physical device, called the Physical Function (PF). Each Virtual Function (VF) has a dedicated hardware resource via a dedicated FP. A single PF can provide a limited number of VFs, often between 8 and 64 VFs. Moreover, despite latency reduction and bandwidth improvements introduced by SR-IOV, bare metal installations still outperform virtualized instances of an application to a large degree [37, 38].

Whiteaker et al.[32] measured the effects of virtualization on network measurements. One of the interesting findings in this work is that sending packets introduces most of the delay while receiving adds almost no additional delay. This is logical as the VM must be executing to send packets, while a suspended VM can have arriving packets buffer at the NIC and then will be immediately available when it next executes. The period of buffering introduces this additional delay.

In order to understand where latency is introduced it is generally necessary to add timestamps to the packets. This can be done in the NIC itself, by the kernel, or in user space by an application. Unfortunately, SR-IOV enabled EC2 instances that have been used in the experiments reported in this thesis do not provide access to the NIC's hardware timestamps.

2.2 Dynamic Replica Selection

This section provides background information to help the reader understand replica selection algorithms. Geo-distributed services are deployed in multiple datacenters across the world and typically maintain multiple copies of their application data. Depending on the data layout employed by the system, this data can be partitioned into shards or tables, such that each logical piece of information is replicated independently.

The term **replication factor** defines the number of times a piece of data is duplicated within a system. This parameter alone does not defines *where* each copy of the data is located. Thus it is possible that multiple copies are stored in a single datacenter or even in a single rack. The replication factor of a particular piece of data is determined by its importance and access patterns. Moreover, this replication factor can be configured statically (via a configuration file) or dynamically (determined by the system at run-time). For example, a social network service can store one copy of the user profile information in the datacenter closest to this user's place of residence, the second replica can be placed in a remote datacenter to increase the survivability of the data, and additional replicas might be placed based on the geographical distribution of the user's friends and followers. The intention of these replica placements is to reduce the network propagation latency associated with the physical distance between the user and the relevant datacenter, thus reducing service response time.

Unfortunately, due to changes in network conditions and uneven load across an application's servers, the closest replica does not always represents the best choice. Luckily, the existence of multiple copies of the data provides a set of alternatives for executing users requests. The term **consistency requirement** determines the number of replicas that must be contacted by the service in order to complete an application request. The consistency requirement of an application request is determined by the consistency model utilized by the service and request's attributes. Naturally, the consistency requirement of a given request cannot exceed the replication factor of the data accessed by the request.

To decide which replicas to contact, latency sensitive geo-replicated data stores, continuously monitor the network and system state among their replicas. These systems process this data and use it to compute the order of replicas based on their *effective* network distance (i.e., as computed based upon delay). The purpose of this ordering is to allow each node to communicate with a subset of the *closest* nodes which are capable of answering queries in the shortest time frame[†].

While we take replica selection as our primary example, the challenges and procedures outlined in this section are generic for a broad set of run-time decisions performed by distributed systems (e.g., replica placement, nearest server selection, quorum formation, etc.).

Figure 2.2 outlines 5 stages of replica selection from the point of view of the replica `R0`, for a cluster containing 6 replicas `R0` to `R5`. The first two steps correspond to `R0` performing measurements and assigning scores to all other replicas. The last three steps are executed on a per-query basis, i.e., given the perceived scores, an algorithm selects a subset of replicas to execute a given query.

[†]In practice, replica selection algorithms can take into consideration the costs associated with contacting a particular datacenter. However, in this work we do not explicitly address this issue.

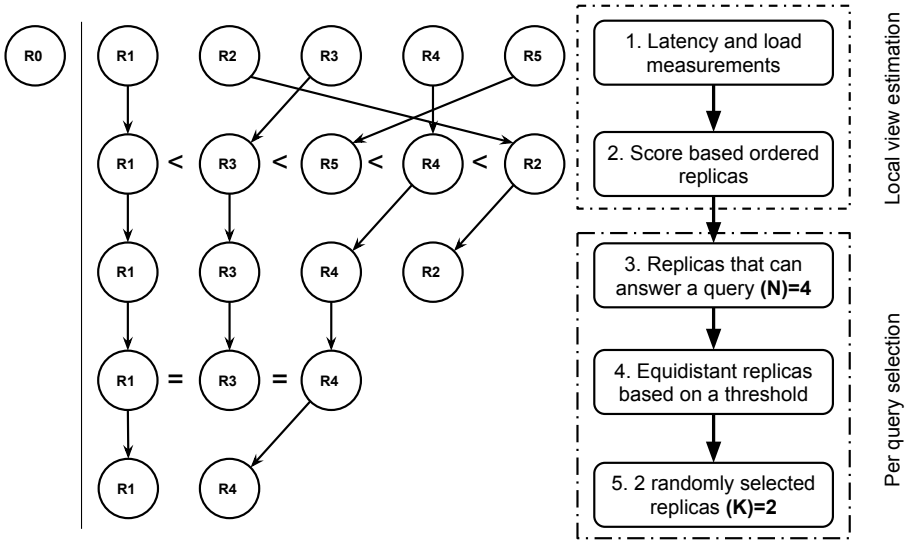


Figure 2.2: Replica R0 performs replica selection from among a cluster of 6 replicas. Out of 5 available replicas, 4 replicas have the data necessary for the specific query. The top 2 replicas are selected out of 4 based on the application's specific logic that desires two additional replicas beyond the copy of the data in R0.

Step 1: Latency and load measurements. The first step includes run-time evaluation of all available replica choices based on an application's specific metrics, such as current network and systems' state. This measurement can be *explicit* or *implicit*.

An explicit latency measurement includes a variation of periodic ping commands executed by R0 towards all other replicas, thus allowing R0 to measure Round Trip Time (RTT) between itself and other machines. Similarly, a periodic enquiry can be performed to retrieve information about the number of outstanding requests and CPU utilization on each replica. The frequency of these measurements typically is a static configuration parameter.

An implicit measurement exploits the existing application requests and responses to infer network latency and the load on the remote machine. This measurement is done by timestamping the application's outgoing requests and incoming responses. The difference between the time when a request has been sent and the time when the corresponding response was received indicates the RTT time in the network and the time it took for the remote replica to process this request[‡]. In contrast to explicit measurements techniques, an implicit approach produces an

[‡]The service time includes the time spent in the Linux networking stack and application service time.

uneven distribution of samples per replica. The number of samples is proportional to the number of requests sent to other replicas (e.g., in Figure 2.2 the number of requests sent from `R0` to `R1` through `R5` is likely to be unevenly distributed). Therefore, the perception of network and systems states of some replicas is more precise than others and this imprecision can negatively affect the quality of the final replica selection (see Section 4.4.3).

Regardless of the sampling technique used, it is common practice to reduce the weights of older samples. This is done to smooth out short term high variance in sampled data and gives greater weight to the most recent samples. Well-known examples of smoothing functions from statistics are the Simple Moving Average (SMA) shown in equation 2.1a and Exponentially Weighted Moving Averages (EWMA) shown in equation 2.1b functions. These functions are commonly used during metric estimation by replica selection algorithms and applied to sampled data, such as RTT or system load.

$$SMA_n = \frac{S_m + S_{m-1} + ... + S_{m-(n-1)}}{N} \tag{2.1a}$$

$$EWMA_n = \alpha \cdot S_n + (1 - \alpha) \cdot S_{n-1} \tag{2.1b}$$

Step 2: Score Based Replica Ordering Individual metrics are integrated into a score which defines the "goodness" of each replica. The score can be formed based upon a single metric (e.g., network latency alone) or a set of metrics. For example, Cassandra uses a combination of end-to-end request completion time and an estimate of the load on remote machines (see Section 4.2.1)). Thus, in the case of Cassandra, the score of each replica is computed as a sum of normalized smoothed end-to-end request completion times to that replica and normalized load that this replica is experiencing at the moment. In contrast, MongoDB uses a simple smoothed RTT as a scoring function.

Replicas are ordered in accordance with their scores. This step is performed periodically at discrete time intervals. The order of replicas is preserved between recomputations and utilized for each query until a new order of replicas is determined. Note, each replica performs independent measurements and constructs its local ordered list of other replicas. Thus, each instance has its own local view of the network delays and loads on other replicas in the cluster.

Step 3: Filter replicas that cannot answer a query. Since the replication factor can be smaller than the total number of replicas in the cluster, not all replicas store the data necessary to perform a certain query. Replicas that do not have a copy of the requested data (thus cannot answer a query) are removed from the list. Thus, the ordered list of replicas is filtered based on the presence of the relevant data in each replica, while preserving the order of the remaining replicas in the list. This filtering is performed frequently, typically once per query.

Steps 4 and 5: Select a final subset of replicas. These steps can be considered optional and are performed only if the set of remaining replicas that can answer the query is *greater* than the consistency requirements of the request. The details of the final replica selection logic are very different between different systems and even between consecutive releases of the same system, thus the following description is an outline of some of the techniques that can be applied.

Two considerations drive this final selection: *replica stability* and *load balancing* [39]. Depending upon the application, a certain cost is associated with a change of replicas (i.e., data might need to be pre-cached at the new location). Therefore, it is desirable to maintain a static set of replicas and thus avoid continuously changing the set of replicas (also known as replica flapping). This is done by introducing hysteresis based upon establishing thresholds that determine the equivalence of replica's scores. Thus, if the difference between the existing order of replicas and the new order of replicas falls below a threshold, the old order is used.

Another approach is to consider load balancing. Load balancing is often done through randomized replica selecting. A set of equivalent replicas (in terms of their scores) is determined based on a threshold, such that all the replicas within that set are considered equivalent regarding their QoS (where we assume that the score is representative of the QoS). Subsequently, a subset[§] of these replicas is chosen at random.

2.3 Symbolic Execution

The main concept behind symbolic execution is to replace concrete input values of an application with symbolic variables rather than numeric values. A symbolic execution engine is used to control the execution process and operates on symbolic variables. Symbolic variables can be seen as placeholders that can change value along an execution path. An execution path is a sequence of choices taken at each branch point in the code. An example of a branching point in C++ is an *if* statement or a *case* switch. The main purpose of symbolic execution is to test and validate applications, by exploring all reachable code paths and automatically generating test cases for all encountered errors.

For each code path within an application, symbolic execution delivers a set of Path Constraints (PCs). This initially empty set is populated with additional constraints at each branch point in the code. The constraint takes the form of a Boolean equation of both symbolic and concrete variables. Automated constraint solvers are used to determine satisfiability of a given PC. The symbolic execution engine maintains a list of all explored code paths and their associated states. The

[§]The size of the subset is determined by the consistency requirements.

```
1  int foo(int X, int Y, int Z){
2      if (X>10){
3        if (X+Y>20){
4          assert(false);
5        } else if (X+Z<15){
6          // UNREACHABLE
7        }
8        return 1;
9      }
10     return 0;
11   }
12   void main(){
13     int X = symbolic();
14     int Y = X+5;
15     int Z = 7;
16     foo(X,Y,Z);
17   }
```

Figure 2.3: Symbolic execution of the function `foo` using symbolic variable `X`.

engine iteratively picks a code path to explore. The duration of the exploration can be time bounded or stop when the end of the program is reached or an error is encountered. The choice of the next state to explore can be randomized or driven by heuristics (for example a depth first search would give priority to code paths with a greater depth).

Concolic execution is a subset of the symbolic execution that allows to mix both symbolic and concrete input variables. The two terms used interchangeably in the rest of this thesis. An interested reader can find more information in [40, 41, 42].

Figure 2.3 shows an example of how a sample function `foo` can be symbolically executed. The function takes three integer variables (`X`, `Y`, and `Z`) as inputs. On the lines 13 to 15 initial arguments are initialized. Variable `X` is initialized to a symbolic variable. Variable Y is expressed as a function of `X` (i.e., `Y=X+5`) and variable `Z` takes a concrete value `7`. Function `foo` (line 1) has 3 branching points and 4 execution paths. The symbolic execution tree corresponding this code example is shown in Figure 2.4.

Upon reaching the first branching point (line 2), the constraint solver evaluates the condition `(X > 10)`. Since the initial PCs set is empty, both execution paths are possible (i.e., where x $>$ 10 and x $<=$ 10). Each code path is assigned its corresponding constraint and then scheduled for exploration according to the search heuristic. Next, the path constraints from the top branching point are propagated down to the following paths. Therefore, during the evaluation of the second condition (line 5) the PC of X $>$ 10 is also taken into consideration.

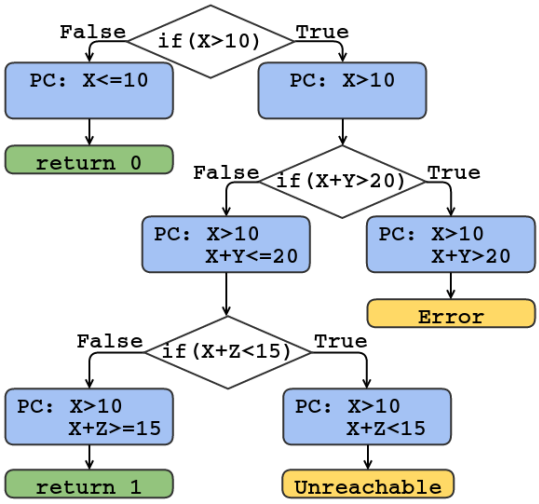


Figure 2.4: Symbolic execution tree, shows all possible code paths for the code listen in Figure 2.3. Path constraints from the higher branching points, propagated down to the bottom of the tree.

When the execution of a single code path reaches its termination point (such as raising an error, assertions, or the end of program), the PC is used to generate a concrete input for every symbolic variable used along that code path. Using the same input will result in the application following the same code path (given that the code is deterministic). Certain PCs may not be satisfiable or impossible to determine due to time limitations or functional limitations of the solver.

One of the main limitations of symbolic execution is state space explosion. The number of possible code paths grows exponentially with the size of the application's source code. This puts a limit on the size of the explored applications and the depth of the exploration that can be feasible to reach. Naive application of symbolic execution will often lead to memory constraint (unexplored states need to be preserved) or the CPU bound (complex conditions involving multiple symbolic variables take time to be evaluated by constraint solvers).

A number of techniques has been developed to address this issue. For example, concolic execution reduces the search space by reducing the number of symbolic variables and replacing them with the concrete values. Reducing the number of symbolic variables also simplifies constraint evaluations. The decision to concretize a subset of input variables can be performed prior to initiation of the symbolic exploration. In Chapter 4 our tool GeoPerf uses domain specific knowledge, to automatically limit the number of symbolic variables.

Despite multiple improvements to date, symbolic execution still remains a challenging process. To efficiently test an application often requires combining a set of optimization techniques with domain knowledge of the problem being explored.

Chapter 3

Latency Measurements in the Cloud

In this chapter* we seek to understand the deployment conditions that geo-distributed systems face when being deployed in a public cloud environment. One of the first questions that we wanted to explore is whether services that are implemented on top of a cloud infrastructure exhibit the same level of latency variability that is observed across the public Internet (in other words, could we statically configure the replicas once and for all?). To answer this question, we conducted a thorough study of application-level and system-level round-trip latencies across all the Amazon EC2 regions. We found that there is significant variability in network latencies over the course of a day, as well as over several weeks. Moreover, there are patterns of behavior that are observable, hence potentially exploitable.

Next, we evaluated how often a replica selection algorithm would experience a different order of replicas. We find that depending on the number of replicas that are being queried, there could be several tens of reorderings during the course of any given day (even when conservatively removing reorderings shorter than 5 minutes). Thus, static replica selection would be suboptimal over a statically configured cloud infrastructure.

Finally, we looked into stability of replica selection by examining the effects of increasing or decreasing the number of replicas.

*The work described in this chapter is based the previous conference paper "The nearest replica can be farther than you think" [1] (the authors of the paper retained the copyright and give their joint approval for this material to appear in this thesis).

3.1 Geo-Distributed Latencies

Today, storage systems are capable of achieving sub millisecond *median* request completion time when serving local requests under normal load conditions. However, when these systems are deployed in a geo-distributed heterogeneous environment, inter-continental network latency becomes the predominant factor in determining the final system's performance. Thus, it is important to understand the dynamics of the network's state.

In this section, we present our network measurements among the geo-distributed datacenters of Amazon EC2. To obtain real world data we instantiated geographically distributed instances of our measurement client (Appendix A.4) on EC2 in all 9 available regions. To date, we have measured latencies between each of these datacenters over a period of 1 year using t2.micro EC2 instances [†]. We collected samples obtained from 3 different echo request/reply (ping) sources: (1) using Internet Control Message Protocol (ICMP) ping, (2) application-level TCP ping (Nagle's algorithm was turned off), and (3) application-level UDP ping. These three different sources of samples helped us to understand the potential causes of latency changes. In our measurements, TCP samples have the highest RTTs due to packet retransmission delays, UDP latencies are equivalent to or lower than the ICMP pings, potentially due to fast path processing of the UDP packets by routers as opposed the sometimes slow path processing of ICMP pings. Thus, we chose the data from measurements of UDP pings as our primary dataset for our further analysis.

Figure 3.1 shows RTT measurements over a period of one day from the EC2 datacenter in Ireland to datacenters in all other regions of EC2. To simplify the visual appearance of this data, we applied a low pass filter to the raw data to remove high frequency noise before plotting the data. Path reordering occurs when two latency curves cross over each other. This figure shows that from the point of view of the Ireland datacenter, the order of paths based on their RTT changes periodically. The importance of any such an event is a function of its duration and the change in the ordering of the top N nodes that are affected by this event. Changing replica ordering too frequently is undesirable, as this may negatively affect caching and other functionality needed for good performance. As a result changing the order of replicates as used by an application should only be done after some period of time. Additionally, the order of the closest replicas has greater impact than a change in the order of the farther replicas. For example consider the top `3` or `top N/2+1` (where `N` is the total number of replicas) that would be the first candidates for a query. The behavior shown in Figure 3.1

[†]Prior to our final measurements, we ran a control test using m3.xlarge instances and found that our limited bandwidth RTT measurement traffic does not seem to be affected by the instance size. All our latency samples are available upon request.

demonstrates that **any** static configuration will perform suboptimally for a large fraction of time. Our detailed analysis of replica selection algorithms used in Cassandra and MongoDB (see Section 4.2), shows that the magnitude of latency changes in our measurements often exceeds default sorting thresholds in the tested replica selection algorithms, thus making these reorderings significant.

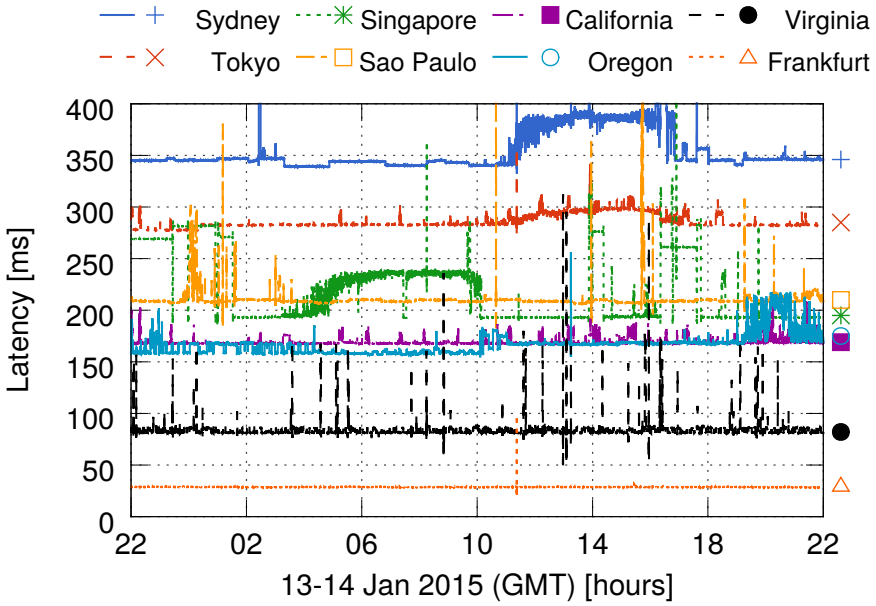


Figure 3.1: RTT measurement over UDP from EC2 Ireland datacenter to all other 8 regions of EC2 (after low pass filter).

3.2 The Global View

To achieve a global view, we tried to answer the question of how often *significant* path reorderings occur across all regions of EC2. To compute the number of reorderings we perform the following steps. **First:** We group paths based on their source region, such that we consider only realistic subsets of paths to order. For example, from the viewpoint of an instance of a latency-aware geo-distributed system, running in the Ireland datacenter, latency is measured from itself to all other 8 regions, while the latencies of the paths between the other replicas are not relevant. **Second:** We applied a low pass filter to our samples in order to remove high frequency noise and high variance (we computed an equivalent to Figure 3.1 for each node). To do this we have converted the latency samples to the frequency

domain using the Fast Fourier Transform (FFT)[‡], and nullified frequencies higher than 1 Hz. Finally, the data was converted back to the time domain using an inverse FFT. **Third:** For each group of links we counted the number of order-changing events and recorded their durations. We ignore events with a duration below a threshold, currently 5 minutes. **Finally:** For each event we identified the indices of the replicas that were affected. The highest index was used to generate Figure 3.2.

Figure 3.2 presents aggregated data showing summaries for two weeks of continuous latency sampling. This figure contains data for different days of the week and different datacenters. The vertical axis shows the number of path reorderings that happened during a day, while the horizontal axis shows the highest index of those paths that have been affected. These plots show how often an application developer can expect the replica order to change based on the geographic location of the datacenter and subset of the closest replicas that are relevant for a specific application. For example, if your datacenter is in California and you perform strong consistency reads implemented by a quorum of nodes ($8/2 + 1 = 5$ nodes) then the median number of reorderings is 15 per day.

3.3 Exploring Replica Position Stability

Next, we explored the stability of replica positions in the nearest-K order from any given datacenter. Once again we applied a low-pass filter to the observed latencies to remove frequencies higher than 1 Hz. We divide the sampling period into intervals of 5 minutes, and at the start of each interval we determine a static replica choice based on the median latencies observed in the previous interval. These choices mimic the behavior of a typical replica selection algorithm. The choice remains static for the duration of the 5 minute window. For each sample, we determine the difference between the static choice and optimal choice at points in time separated by 1 second. At the end of the interval, we output the median and maximum difference encountered during that interval. This difference estimates the time one would lose by retaining a static replica set chosen based on the past interval's performance. We repeat these computations to cover all possible Top-K combinations, considering the delay between the datacenter at hand and its closest K replicas. For example, Top-2 refers to the case of a total of three replicas.

Figure 3.3 contains two sets of graphs showing the median and maximum delays observed from the Ireland and Virginia datacenters. We chose these datacenters because they are the closest ones to large fractions of users in Europe and the Eastern US. The results are surprising. For example, having only two additional

[‡]We assume that the data is periodic and not related to the specific time interval of our observations.

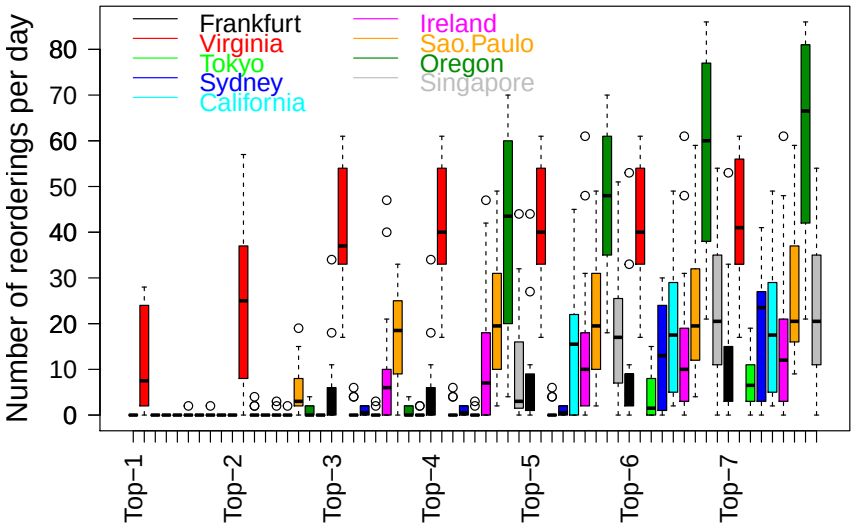
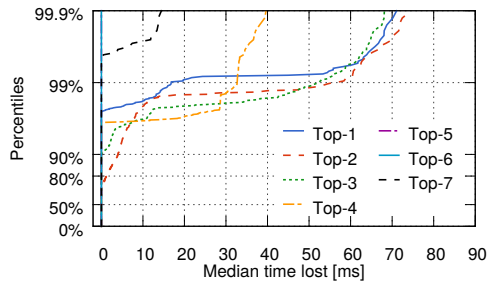


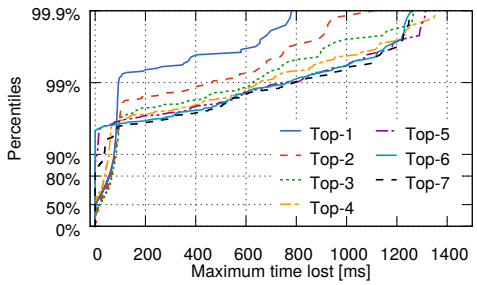
Figure 3.2: Change of order for the closest K out of total 8 nodes, per region per day over 2 weeks. The color of the boxes corresponds to different EC2 regions. 14 days are aggregated into one boxplot, where the top and the bottom of each box indicate 25th and 75th percentiles, outliers are represented by circles. The vertical axis indicates the number of reorderings that happened on a particular day on a particular datacenter. The horizontal axis indicates the highest indexes of the top K nodes affected. Top-8 reorderings are identical to the Top-7 and thus not shown in the graph.

replicas dramatically increases the uncertainty and variance as observed from the Virginia datacenter, as shown by the significantly greater number of time intervals affected (Y-axis) and median time lost (X-axis) in Figure 3.3a. In particular, more than 29% of time intervals suffer some median time lost, up from 3% for Top-1. Adding additional replicas can decrease the uncertainty, for example Top-5 shows only 0.1% of time intervals with time lost. This behavior is directly driven by the variability of the paths from the datacenter being considered and the other replicas. For example, Top-7 replicas again increases the variance to 0.3% of affected time intervals. However, adding additional replicas makes the selection process more vulnerable to unexpected delays and this increases the maximum time lost, as shown in Figure 3.3b. This figure also shows that roughly 1% of time intervals experience latency penalty of about 600 ms. This is at least an order of magnitude greater penalty than the median time lost (Figure 3.3a).

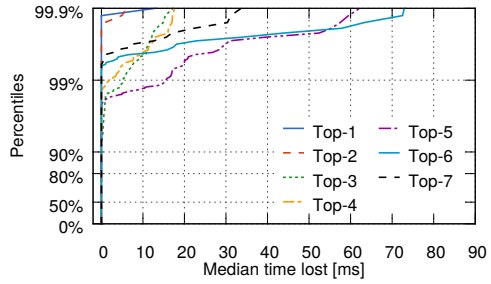
The view from the Ireland datacenter is qualitatively the same (Figure 3.3c), but the unstable replica positions and the times lost differ. Here, Top-4 replicas show fairly low uncertainty with 1.4% of time intervals affected. However,



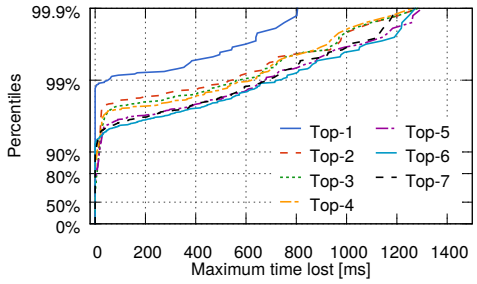
(a) Median time lost from Virginia



(b) Maximum time lost from Virginia



(c) Median time lost from Ireland



(d) Maximum time lost from Ireland

Figure 3.3: Median and maximum time wasted for the window size of 5 min from Ireland and Virginia. Including more replicas typically increases the maximum penalty, but can produce more stability by going beyond replica positions with high variance. Top-8 configurations use all available replicas and by default perform optimally and thus not shown here.

uncertainty is harder to eliminate by more than a factor of 10; with close to 1.5% of intervals showing some median time lost for Top-4, as opposed to 0.1% for Top-5 from Virginia. In contrast, Top-5 from Ireland shows the worst variance.

These findings demonstrate: (i) the need for careful adaptation by the replica selection mechanisms and (ii) the difficulty in producing robust algorithms that work well across a variety of network conditions.

3.4 Summary

In this work we demonstrate the need for dynamic replica selection within a geo-distributed environment on a public cloud. We conducted thorough round-trip time measurements across *all* geo-distributed datacenters belonging to one cloud provider (Amazon EC2), for periods ranging from several weeks to a year. To the best of our knowledge, no such measurements are publicly available. Using this data, we show that the replica orderings would change up to several tens of times per day, from any given datacenter's viewpoint. Also, we explored the

stability of replica positions in the nearest-K order from any given datacenter and find a surprising amount of variability. We applied low-pass filtering to eliminate the noise in the latency measurements caused by the server virtualization platform. Even after this filtering, it is evident that there are substantial latency variations across this provider's wide-area network.

Chapter 4

Systematic Testing of Replica Selection Algorithms using GeoPerf

In the previous chapter, we described our latency measurements across Amazon EC2. These measurements showed high variability in network conditions and changes in the best set of replicas over time. In this chapter*, we look closely at the two implementations of replica selection algorithms used in two popular storage systems Cassandra and MongoDB. We begin by highlighting the complexity of testing and verifying such algorithms, particularly in the presence of changing network conditions, as demonstrated in the previous chapter.

To overcome this complexity and validate replica selection algorithms we propose GeoPerf, a novel tool that tries to automate the process of systematically testing the performance of replica selection algorithms for geo-distributed storage systems. The key idea behind GeoPerf is to combine symbolic execution and lightweight modeling to generate a set of inputs that can expose weaknesses in replica selection.

We tested Cassandra and MongoDB using our tool and found bugs in each of these systems. We use the latency traces described in Chapter 3 both as (1) domain specific knowledge in GeoPerf's heuristic to mitigate state space explosion of the symbolic execution and (2) to quantify the time lost due to these bugs.

*The work described in this chapter is based on the conference paper "The nearest replica can be farther than you think" [1] (the authors of the paper retained the copyright and give their joint approval for this material to appear in this thesis).

4.1 Introduction

Achieving high throughput *and* low latency responses to client requests is a difficult problem for cloud services. Depending upon the replication policies, the consistency model of a service, and the current network state, clients have to choose which replica or set of replicas they will access, using replica selection algorithms (see Section 2.2). These algorithms are expected to consistently make excellent choices despite the unstable environment of geo-distributed systems spread across the globe.

The replica selection process is inherently hard. The service's clients may conduct both passive and active measurements of the latency of the served requests, then use this past history to drive future choices of which replicas to use. Unfortunately, competing traffic over the links where the requests and responses are propagating is bursty, and routing decisions (that often do not take the overall network's performance into account) frequently changes as well. As a result, end-to-end network bandwidth, latency, and loss rate may change dramatically across the wide-area network. To try to compensate for these issues, a replica selection process needs to include mechanisms for filtering and estimating the likely latency of each request when deciding how to process requests. These mechanisms need to answer a number of difficult questions, for example: (i) For how long should the past latency samples be kept? (ii) What should be the adaptation rate? (For example should new latency samples be given preference?) (iii) How to deal with highly variable samples, e.g., discard some outliers or pay attention to them?

Unfortunately, errors (suboptimal choices) in replica selection algorithms are extremely hard to find. One of the things that makes such errors difficult to detect is that these errors usually do *not* result in critical system failures. Moreover, it is hard to determine the system's optimal behavior in the absence of up-to-date, full global knowledge. Thus we either have to know what the optimal behavior is or we need full global knowledge to compute what this behavior should be in order to detect the presence of an error. Suboptimal replica choices can result in increased latency, and the poor user experience due to these suboptimal choices can drive a significant fraction of the customers away [20]. Thus, it is important that requests are served by replicas that provide the client with a response within the expected (bounded) time, and the better the selection algorithm the greater the reduction in latency (until the selection is in fact optimal).

Debugging replica selection algorithms is difficult for a number of reasons. Bugs can occur due to sampling problems, in calculations, in selection logic, etc. To thoroughly test the selection algorithms we would need to cover all possible network topologies and their bandwidth, latency, and loss rate characteristics. In addition, we would need to anticipate the exact intensity, duration, and frequency

of changes in traffic and routing, which is difficult[†]. Thus, it is unlikely that unit tests and simple (no matter how long) simulations can identify all the bugs. Instead, a *systematic testing tool* is needed.

In this chapter we describe GeoPerf, our tool for automating the process of testing replica selection algorithms. We choose to apply symbolic execution [43, 44], because it systematically uses the code itself to identify test inputs that can cause the code under test to execute all branches in the code and ultimately traverse all possible code paths. In our case the inputs are the changing latencies presented to the replica selection mechanisms. As a result, we use the symbolic execution engine to answer the difficult aforementioned questions and systematically look for bugs. However, using symbolic execution comes with its own set of challenges. **First**, detecting errors using symbolic execution requires an application to violate certain invariants, such as an assertion in the code. Specifying these assertions is trivial in the case of memory bugs. Unfortunately, in our case we are dealing with performance deviations and we need the ground truth to define what a violation would be. **Second**, if the symbolic execution engine were to propose inserting latencies that go significantly beyond the observable ones, one could question the relevance of the bugs found. Thus we need to utilize realistic latencies. **Third**, it is easy to run into a path explosion problem due to the large number of possible branches in the code. A large number of symbolic inputs can similarly cause an exponential explosion in state space. Unfortunately, most of the replica selection algorithms contain filtering mechanisms that would require several symbolic variables should symbolic execution be applied blindly.

To provide the ground truth, we use lightweight modeling to approximate what an optimal choice of a replica should be. Our ground truth is a straightforward selection of closest replica(s) based on latencies smoothed-out in the same way as in the system under test[‡]. This means that the code under test and our model are fed the same latencies in a discrete event simulator, and symbolic execution then tries to find the inputs for the next iteration that would cause a divergence in the choice of replica(s). Any such case is a potential bug. To address the second challenge, we simply reuse the latencies we collected across the WAN when running over Amazon EC2 geo-distributed datacenters, and use them to limit the input ranges that the symbolic execution engine is allowed to propose (see Chapter 3). Finally, to address the third challenge we apply domain-specific knowledge regarding the way the various latency filtering mechanisms work, enabling the symbolic execution engine to work across several iterations. In particular, we make it possible to

[†]If it were easy to do this, then we would already be using the perfect replica selection algorithm!

[‡]Unless that module is suspected to be buggy, in which case we change the smoothing mechanism that we use for ground truth.

use only a few symbolic inputs to drive execution along different code paths and produce results in a matter of hours.

GeoPerf's contributions are listed in Section 1.3.

4.2 Systems That Use Replica Selection

While many different systems utilize replica selection, in this work we have focused on two:

Cassandra is a highly configurable NoSQL distributed database, designed to work with large datasets in local and geo-distributed environments. Unlike many other distributed databases, Cassandra can perform read operations with a per request variable consistency level, depending on the client's requirements. The client communicates with one of the replicas (presumably the one closest to the client). This node selects a subset of closest replicas to itself (including itself), forwards the client's request to them and waits for their replies. Once enough replicas have replied, a single reply is sent back to the client. The consistency level can vary from LOCAL (i.e., reads from a single node), QUORUM (i.e., $N/2 + 1$ replicas), to ALL where all nodes have to respond prior to returning an answer to the client.

Generally, a configurable number of nodes will be used to produce an answer, thus allowing a tradeoff between consistency, availability, and performance – as acceptable to the client. While performing a quorum read is well understood, being able to choose a specific subset of nodes opens a number of possibilities, especially if these nodes (replicas) are geo-distributed.

MongoDB is a popular distributed document management database. It supports atomic, strongly consistent write operations on a per document basis and either strong or eventually consistent read operations depending on the client's preferences. MongoDB implements a master-slave replication strategy, with one primary and a number of secondary nodes. All write operations are directed towards the primary node and eventually propagated to the secondary replicas. Replication increases redundancy, data availability, and read throughput for clients that accept eventual consistency semantics. By default, a client's read operations are directed to the primary machine and return strongly consistent data, although the client has an option to use secondary replicas, or to choose the closest node regardless of its current status. We concentrate on the case of choosing the closest node.

4.2.1 Replica Selection Algorithms

Replica selection systems usually contain two elements: a smoothing algorithm (filter) to get rid of high variability, and the actual replica selection algorithm that acts based on the latencies output by the filter.

For latency smoothing, Cassandra uses the off-the-shelf Java Metrics library v2.2.0 [45]. This library implements a special type of a time-decaying function called Forward Decay Sample[46] (via the library class Exponentially Decaying Reservoir (EDR)). Similar to Exponentially Weighted Moving Average (EWMA) it gives greater weight to recent samples.

Latency estimation in MongoDB is the responsibility of the client's driver, which chooses the closest server to connect to. MongoDB has multiple clients for compatibility with many programming languages. Currently there are more than a dozen different client drivers (including open source community drivers). Naturally many of them have been implemented by different developers, thus their implementations have significant differences in peer selection. Table 4.1 lists the subset of these drivers that we examined. The table also shows that the choice of latency smoothing is heterogeneous. In our work, we concentrate on comparing the C++ and Java drivers, as these are considered to be the most widely used and also represent the most distinct implementations in terms of latency smoothing. Note that the C++ driver uses EWMA with a coefficient of 0.25 applied to new samples, while the Java driver uses a Cumulative Moving Average (CMA) (arithmetic average across all collected values). [§]

Replica selection algorithm Cassandra implements a module called *Snitch* to help each node choose the best set of replicas to which to direct read operations. There are several types of Snitches [47] that allow administrators to tailor the logic to the deployment environment. For example, for multi-region geo-distributed deployment, each replica is associated with a distinct region and distinct rack within a datacenter. This allows Cassandra's nodes to be grouped based on their physical proximity. This is a static configuration, while in this work we are more interested in the Dynamic Snitching implementation. *Dynamic Snitch* automatically chooses the closest node(s) based on the network distance and node load, with both parameters incorporated into the score function. In particular, *Dynamic Snitch* automatically chooses the closest node(s) as follows: (1) Asynchronously, each replica contacts every other replica with request messages. The time it took from the request until the reply is received is passed through the EDR smoothing function. This time includes both the network component (i.e., RTT) and the retrieval (service) time at the remote node (the later is an indirect indicator of the load at that node). (2) Periodically, current

[§]In the recent version of the Java Driver (since 3.0.0) the logic has been changed to an EWMA smoothing function with a coefficient 0.2.

Table 4.1: MongoDB's smoothing functions from different drivers. The drivers' source code obtained from https://docs.mongodb.org/ecosystem/drivers/.

Driver Version	Latency Filter Type	Maintenance
C++	EWMA(0.25)	official
Python	EWMA(0.20)	official
Java	CMA	official
C	EWMA(0.20)	official
C Sharp	EWMA(0.20)	official
Perl	EWMA(0.20)	official
Ruby	EWMA(0.20)	official
Node.js	Not identified	official
PHP	Not identified	official
Go	Maximum of the last 6 samples	community
Erlang	EWMA(0.20)	community

values from the smoothing filters are collected and normalized, providing a list of scores assigned to each replica. By default, this process is executed every 100 ms. (3) These scores are used when the local node needs to forward client requests to other replicas. The selection process itself is executed in two stages. First, all replicas are sorted based on their physical location, so that all replicas in the same rack and then in the same datacenter as the source are at the top of the list. Second, the delta of a score is computed from the local node (originator of the query) to all other nodes. If this delta is greater than a threshold (default configuration is 10%) of the difference to the closest node, then all nodes are sorted based on their score. Finally, the top K elements from the list are chosen[¶].

Dynamic Snitch uses client requests to extract information about the state of the network and load on other replicas. The downside of this approach is that not all replicas are sampled evenly. Slowly performing replicas serve fewer client requests; thus, they have few latency samples. Therefore, their perceived performance will not accurately represent the actual state of the network. To address this issue, Dynamic Snitch resets all collected samples every 10 minutes (default configuration). This equalizes scores among all replicas and allows Cassandra to periodically send a small fraction of requests to all replicas, thus giving them a fair chance to regain their score. However, this results in periodic poor selection of replicas.

Unlike Cassandra, MongoDB drivers (both C++ and Java) rely only on the RTT to compute network distance to replicas: (1) The collected latency samples

[¶]Based on Cassandra V2.0.5.

are passed through EWMA with the filter coefficient 0.25 and CMA in C++ and Java drivers, respectively. The sampling occurs at a hard-coded heartbeat frequency of 5 s in both drivers. (2) Upon the client's request, the smoothed latencies are used to sort all relevant replicas (i.e., generating a replica set that can answer a query). All replicas that are farther than the default threshold of 15 ms from the closest are filtered out of the list. (3) Finally, one replica is selected at random from the remaining list.

Both systems address issues of caching, by avoiding unnecessary switching between replicas under high latency variance. Secondly, both systems utilize randomness in making their choices, this has a positive effect on balancing workload and helps to prevent a herd effect (when all nodes simultaneously go to the closest node, both congesting and overloading a node [39]).

4.3 GeoPerf

In this section, we describe our approach to systematically test replica selection algorithms. First, we isolate the core logic of the replica selection algorithm. We can use our light weight modeling as the implementation of the ground truth or a different algorithm as a reference. Then, we systematically examine code paths in the original algorithm in an effort to find a case in which the algorithm under test performs worse than the reference one. For path exploration, we use symbolic execution, a common technique for software testing. It incorporates systematic code path exploration with an automatic constraint solver to derive concrete input values that will cause a particular code path within a program to be executed. In our case, the inputs are the latencies that could be observed by the replica selection algorithm under test.

4.3.1 Systematically Exploring Replica Selection Algorithms

We use the popular open-source symbolic execution engine KLEE[44] with the STP[48] constraint solver. While KLEE (and symbolic execution tools in general) is primarily designed as a test generation tool, it can also be applied to a great variety of other problems. Its ability to generate test cases that satisfy all the constraints can be seen as a solution to a problem defined through code. For example, by symbolically executing code that has a branching point of the form `if ((X+5)*(X-2) == 0){assert(0);}` where `X` is a symbolic variable, the equation will be passed on and solved by an automatic constraint solver. The generated test case will be a solution to the quadratic equation.

Felipe Andres Manzano, in his tutorial[49], demonstrates how KLEE can find an exit out of a classical maze problem by exploring a set of inputs that lead to distinct code paths, and eventually the exit out of the maze. If a program contains a code path that would solve a particular problem, then our task is to define the problem in a clear way so that the symbolic execution tool can isolate a code path (or a set of code paths) that leads to the desired state.

The choice made by the replica selection algorithm depends on the system state (i.e., past and current set of network RTTs, time, random numbers) and the algorithm itself as implemented via code. Each potential replica choice is defined by a set of possible code paths that can lead to it. In the compiled programs only a single code path will be executed, as determined by the system state at each branch point. However, by symbolically executing this code we can explore alternative code paths that would lead to different choices given the symbolic state of the system. Therefore, for each alternative code path we can determine a set of constraints, and the automated theorem solver will derive a state that would result in the alternative execution path. In other words, we can say what would be the latency over the network paths to cause a replica selection algorithm to pick one or another subset of nodes.

Forcing a replica selection algorithm to make different choices is insufficient to reason about the correctness of that algorithm. To do this, we need to know the ground truth. Alternatively another algorithm can take this role, allowing us to compare two choices in a design exploration.

GeoPerf compares a pair of replica selection algorithms by using symbolic execution and lightweight modeling. It symbolically executes them in a controlled environment, while both algorithms share one view of the network. We use symbolic latencies to check if one of the two algorithms makes a different choice under exactly the same conditions.

However, there are several limitations that need to be addressed. First, symbolically executing an entire distributed system is still difficult and requires inside knowledge of the system, code modifications, and significant computation resources. Unfortunately, this would introduce a lot of code paths that are irrelevant to the replica selection logic. Second, symbolic execution does not have a notion of continuous time, which makes it hard to evaluate replica selection choices. Finally, we want to develop a general purpose tool (a common platform), independent of a single distributed system and suitable for quick prototyping and testing of various algorithms, potentially different versions of a system or even completely independent solutions.

For all of the above reasons, we isolate the replica selection algorithm from the systems under test, and develop a controlled environment where we can deterministically generate, monitor, and replay events as needed.

```
1   void main(){
2     Sim A = Sim("algorithmA");
3     Sim B = Sim("algorithmB");
4
5     A.run("RTTs.data");  // Preload initial states into both
6     B.run("RTTs.data");  // simulations
7
8     // Declare new symbolic variables
9     int sym_lat = symbolic[nodes][depth];
10
11    timeA = A.run(sym_lat);
12    timeB = B.run(sym_lat);
13
14    assert(timeA < timeB);
15 }
```

Figure 4.1: Event based simulation pseudocode

The core of the tool is based on our own discrete event-based simulator developed as part of GeoPerf. The setup simulates: (i) a set of geo-distributed nodes connected via WAN paths, (ii) arrival of incoming client requests, and (iii) a replica selection module that periodically chooses a subset of nodes to serve these requests.

We create two instantiations of the simulator, one using the reference replica selection algorithm and the other the algorithm under test. Both instances run in parallel in identical environments (using synchronized clocks and deterministic synchronized pseudo-random number generators). KLEE is used to drive the exploration of the code paths generating a set of symbolic latencies (i.e., inputs) that characterize the network paths among the nodes. The target of the exploration is to find a sequence of network states that exposes potential weaknesses (bugs) of one of the algorithms by repeatedly demonstrating inferior performance (choices) in the simulated environment.

4.3.2 Comparing the Selection Algorithms

Consider the code example in Figure 4.1. At lines 2-3 we create two simulation instances that differ only in the algorithm used for replica selection. On lines 5-6 we preload initial states into both of these simulations by replaying identical sets of latency inputs that we collected earlier. Line 9 declares a set of symbolic variables to be used as an input to future iterations of these two simulations. Lines 11-12 run both simulations with the new symbolic input and collect accumulated request times. Finally, on line 14 we have an assertion that will be triggered when the estimated performance of the simulation guided by algorithm B is slower than its

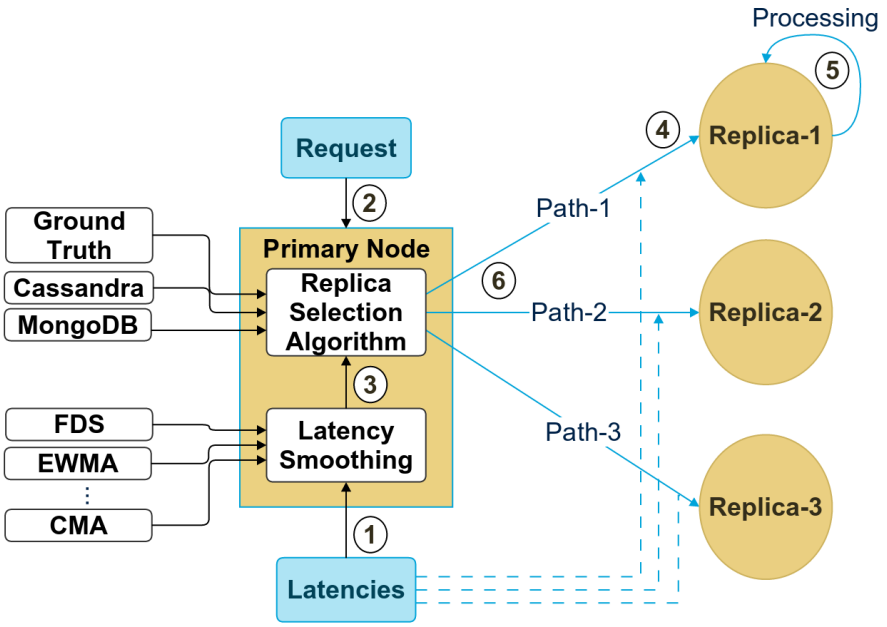


Figure 4.2: Discrete event based simulation: (1) latencies assigned to inter replica paths and passed through the smoothing filter, (2) client's request generated, (3) the replica selection algorithm is used to chose a closest replica(s) to forward the request, (4) request forwarded to the replica(s) (5) replica processing the request (6) the reply sent to the originating node.

counterpart A. At this point KLEE will have the set of PCs that led to this state, obtained after exploring the code paths of both simulations.

Next, an automated theorem prover will determine if it is possible to trigger the assertion given the current set of path conditions. If so then we can generate a concrete test case that will cause this difference in performance. The concrete values calculated for the symbolic variables are appended to the file "RTTs.data" and will be used during the set-up of the next iteration of the tool.

The pair of algorithms (A and B) are the models of the algorithms from the systems that we test or model. In the process of modeling, we isolate the logic behind the algorithm's implementation and replicate it line by line for our tool. The actual size of such modules is quite small, less than 200 lines of code in both Cassandra's dynamic snitch and MongoDB's drivers.

Ground truth To provide the ground truth when the counterpart algorithm B is unavailable, we generated a simplified replica selection mechanism that always selects the node with the lowest delay, without randomization or optimization thresholds. Ground truth is an approximation of the optimal performance and it represents the minimum bound in the achievable latency, provided that there is no caching of requests. We expect that algorithms under test will demonstrate

lower performance than the ground truth in certain cases, but will asymptotically produce comparable results. By default, ground truth uses the same latency smoothing function as the algorithm under test. If the smoothing function itself is suspected to be buggy, it can be replaced.

4.3.3 Discrete Event Simulation

As a platform for our evaluation we created a discrete event simulator. This simulation models a set of three interconnected geo-distributed replicas and one additional primary node that serves requests (Figure 4.2). We feed to the simulation client requests generated at a constant rate. These requests are forwarded to the primary node in the set. The primary node runs a replica selection algorithm to choose to which replicas it should forward each request. The other nodes in the set are potential replica candidates; they receive the forwarded requests from the primary node, serve them, and send replies back. We also input the set of network latencies that describe RTTs on each path from the primary node to all the replicas. These latencies represent the time it takes for requests to reach the replicas and for the replies to return. As the simulation time progresses, new latency values are introduced to reflect the changing network conditions.

The raw latencies are fed into a smoothing function (such as EDR, EWMA, or CMA) according to the specific system being simulated. The output of these functions produces the perceived latency that is used within the replica selection algorithms. Each algorithm makes choices periodically (100 ms for Cassandra and 5 s for MongoDB as per their default configuration). All requests received in that time interval are directed towards the replica set selected at the last decision time. The time it takes to process individual requests is determined by
$T_{total} = \frac{RTT_{request}}{2} + T_{processing} + \frac{RTT_{reply}}{2}$

We do not consider request queuing and use a fixed delay of 0.5 ms to generate a response at the replica, i.e., a fixed service time. We obtained this service delay experimentally by running a real system on our hardware [‖], without any load. Such a choice is further motivated by the fact that the processing time is already incorporated into the Cassandra logic, and is completely ignored by the MongoDB logic (at least in those drivers for replica selection we tested).

4.3.4 Iterative Search

Ideally, we want to ask the symbolic execution engine to compute the behavior of the system hours ahead of time, and to tell us what sequence of events and inputs could result in undesired behavior. However, one of the biggest challenges in

[‖]Intel Xeon E5-2670 clocked at 2.6 GHz with 96 GB of DDR3-1600 MHz RAM.

symbolic execution is the path explosion problem. The number of possible paths grows exponentially and the exploration eventually faces a bottleneck, such as memory or CPU time bounds. The number of possible code paths is exponential in the number of symbolic variables used in the exploration. This becomes the predominant factor that limits the maximum number of symbolic latency variables that we could practically use in our exploration to 9 (3 triplets, i.e., 3 latency inputs for the 3 replicas in the simulation). The number of triplets defines the depth of the exploration, as each consecutive latency input describes network conditions within the simulation. It is important to note that the number of code paths is also dependent upon the complexity of the algorithm under test. EDR, used in Cassandra's logic, has many more code paths than a simple CMA or EWMA. A single latency triplet input (or a single iteration) corresponds to 100 ms and 5 s of simulated time for Cassandra and MongoDB respectively, as configured in these systems by default.

However, it is insufficient to show that two algorithms make different choices at a single point in time. It is important to show that this choice exhibits a pathological behavior that lasts long enough to cause significant performance degradation for many requests. Thus, we apply an iterative approach by repeating individual explorations.

There are two distinct states to consider. First, is the state of the events in the simulation, which describes the queue of the discrete events (e.g., the messages in transition). This state is used to evaluate the performance of the system. Second, the state of the history buffer in the latency smoother is used in the algorithm, for example Cassandra's EDR is configured to remember 100 previous samples. These states may differ from the start of the simulation due to different smoothing filters.

Figure 4.3 outlines the high level idea behind iterative search. (1) We configure two simulations with a pair of distinct replica selection algorithms, and warm up the system by inserting a set of previously measured latencies to pre-populate the history buffers used by the latency smoothing functions. At this stage, both simulations are in the same state, as replica selection algorithms make identical choices. However, since the smoothing techniques can be different (i.e., CMA and EWMA), their states differ as well. Warming up the simulation with concrete values does not lead to performance penalties due to the Execution-Generated Testing (EGT) [50] technique implemented in KLEE. EGT allows us to distinguish between concrete and symbolic variables, and avoids creating path constraints if no symbolic variables are involved. (2) As input we introduce a set of symbolic latency triplets, and initiate symbolic exploration. GeoPerf continues the simulation and starts to acquire PCs for all explored paths. After inserting the last symbolic input and finishing all outstanding queries, the performance of both simulations is

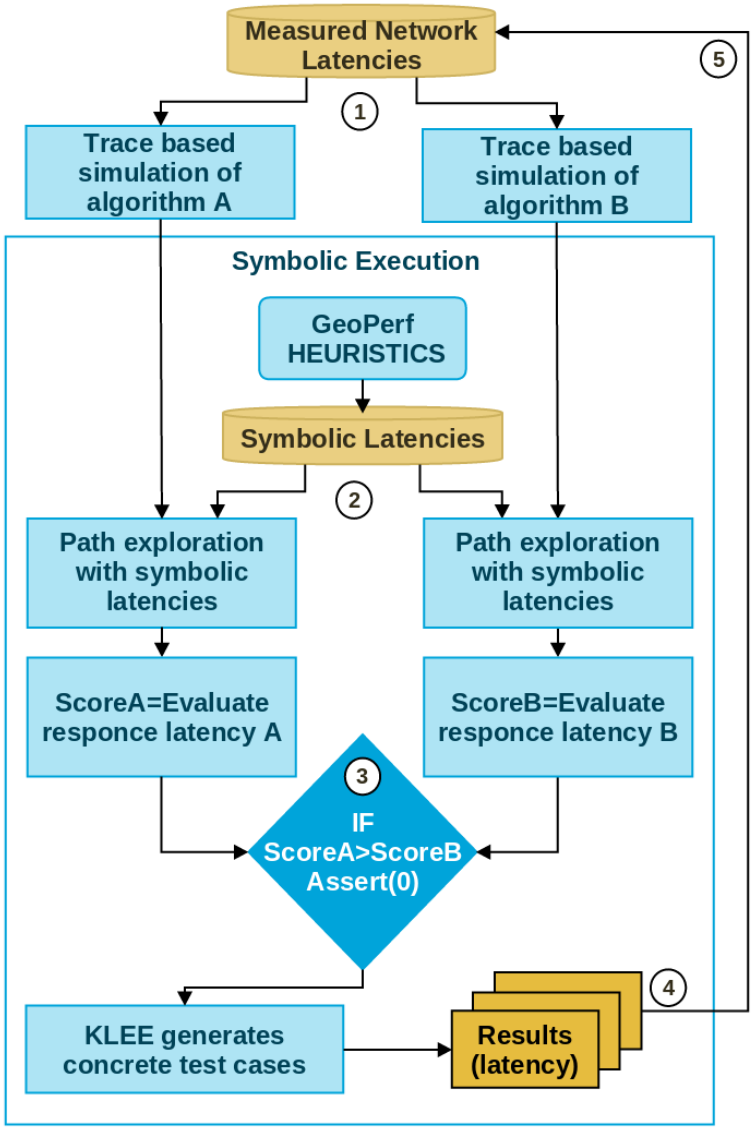


Figure 4.3: GeoPerf Overview

evaluated. As the scoring function we consider the accumulated time to generate requests initiated during the insertion of symbolic inputs.

At stage (3) we compare the scores. Consequently in the code, we have an assertion checking that the total accumulated time of simulation-1 should be more than the accumulated time of simulation-2. The PC is sent to the constraint solver and checked for satisfiability given the current code path. If the PC is not satisfiable, then we continue searching until we reach the assertion point through an alternative code path or until we run out of possible code paths, meaning that

regardless of the state of the network latencies it is impossible to manipulate the two replica selection algorithms into making different decisions. The continuation uses a different configuration of the search as described in Section 4.3.5. Finally, in the successful case when we reach an assertion, KLEE uses the constraint solver and the obtained PC to generate a concrete test case (i.e., convert symbolic triplets into a concrete set of latencies) that deterministically bring the simulation to the same assertion and therefore replica selection choices diverge.

At stage (4) we pick a newly generated set of latency triplets and append it to our initial set of EC2 latencies. At stage (5) we restart our simulation from the initial point. However, in addition to the original set of measured latencies, we also replay the latencies generated in the previous iteration as concrete input.

4.3.5 Optimizations

We have developed a heuristic technique to guide the symbolic execution in an effort to find the desired sequence of latency samples. There are three configuration parameters that define the search: maximum search time, number of solutions to find, and number of symbolic variables to use. For each iteration we attempt to find a subset of solutions within the given time limit. If by the end of its time this set is not empty, we sort all our solutions by the improvement ratio that they introduce. However, if we have run out of time or possible code paths, we then attempt to use a different symbolic pattern (SP). In our work we define s SP as a set of assumptions (relationships) between the symbolic variables that we input to our simulation. We use the observations of measured samples and [51], to configure the SP for each iteration of the search. One observation is that there are different periods of time during which latency does not change significantly (thus we can treat this period of time as if the latency does not actually change). Another observation is that typically only one link will change by a large amount at any given time. By following these observations, we can use a single symbolic variable to define path latency over several iterations of a simulation. Moreover, we can leave the latency of some paths unchanged by reusing values generated in the previous iteration of the search. This technique reduces number of symbolic variables used per iteration and speeds up KLEE's code path exploration by simplifying PCs and reducing the code path search space.

When our exploration reaches the point when it cannot find a solution with a given configuration, we systematically pick different SPs by re-using symbolic variables over several iterations of the simulation. This optimization allows us to increase the depth of search up to 10 replica selection choices ahead.

In addition, our experiments show that using integer numbers significantly speeds up constraint solving. Therefore, we replaced floating point operations with their integer equivalent, three digits of precision.

4.4 Evaluation

In this evaluation we describe the bugs we found (Section 4.4.1). Then, using our latency traces obtained in Chapter 3 we quantify the bugs' impact (Section 4.4.2). Finally, in Section 4.4.3 we use GeoPerf to identify effects of periodic buffer resets on quality of replica selection, then using latency traces we evaluate it's impact.

4.4.1 Bugs Found

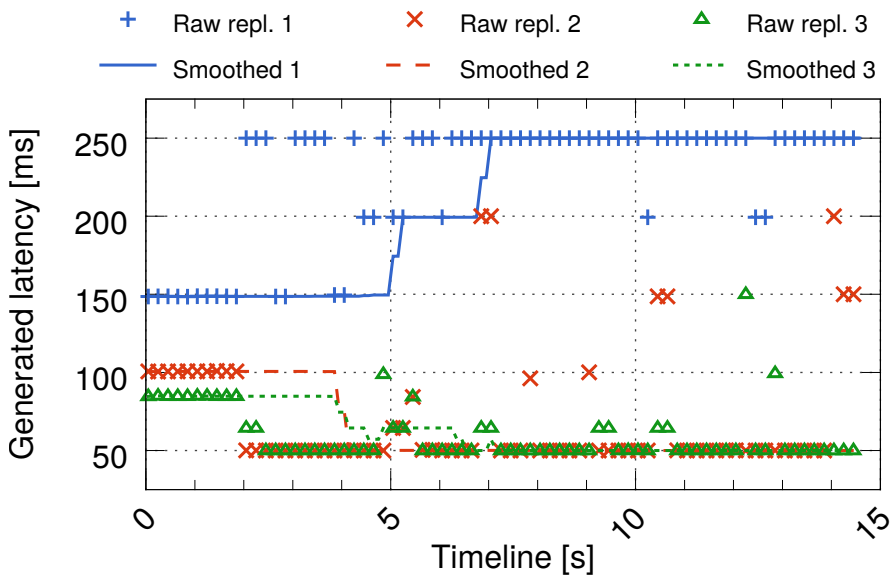
Cassandra First, we use GeoPerf to evaluate the performance of the Cassandra's Dynamic Snitch against the ground truth model provided with GeoPerf. Both algorithms were configured to use EDR as the latency smoothing function. GeoPerf's simulations were configured to sample latency at 100 ms intervals. All GeoPerf's explorations were preloaded with 20 samples from EC2 measurements per network path. The RTT range for symbolic latencies was set to 100-500 ms to represent an entire range of observed latencies within EC2. We use a fixed request arrival rate of 40 requests per second.

It was sufficient to run GeoPerf for 130 iterations (equivalent to 15 s of real-time and 390 symbolic latency inputs) to notice the problem. Figure 4.4a shows the latencies as generated by GeoPerf and the smoothed version after passing through EDR. This figure demonstrates that smoothed EDR latencies follow the general trend and closely represent the real state of the network. However, Figure 4.4b shows that despite having the correct view of the network, over 20% of Cassandra's requests have RTTs of 500 ms (they are forwarded to the slowest node of the three). This clearly indicates an issue in the replica selection logic. By examining the code, we have identified that the problem was caused by a bug in the replica score comparison function as follows:

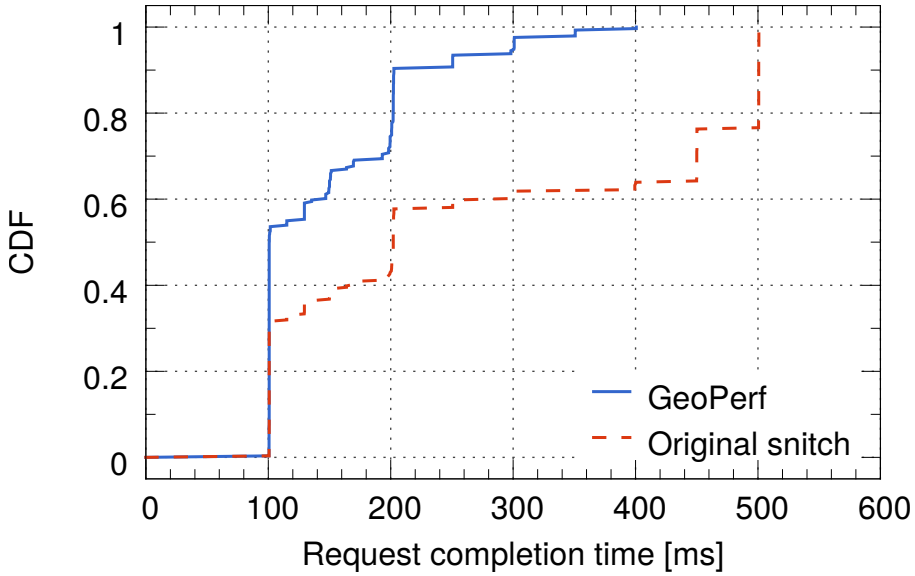
```
if ((first-next)/first>BADNESS_THRESHOLD).
```

As the value of `next` score could be greater than the `first` score, this results in comparison of the threshold with a negative value. Ultimately, this prevents the sorting function from being called.

MongoDB We compared the performance of the C++ and Java drivers of MongoDB. Their only difference is the smoothing function used: EWMA (0.25) and CMA for C++ and Java version respectively. In contrast to Cassandra's exploration, GeoPerf was now configured to compare the two provided algorithms (the use of ground truth was unnecessary in this case). The system was set to use a 5 second sampling interval to match MongoDB's logic. We ran a simulation for a total of 230 iterations (equivalent to 20 minutes of real-time and 690 symbolic entries). Figure 4.5a shows the latencies generated by GeoPerf and those perceived by MongoDB logic after passing through the Java driver's smoothing function. The latencies generated by GeoPerf demonstrate the inability of the Java driver



(a) Set of 3 path latencies generated by GeoPerf and the smoothed version of these latencies after FDS



(b) CDF request completion time, Cassandra's Dynamic Snitch compared with GeoPerf's ground truth algorithm

Figure 4.4: Comparing Cassandra's Dynamic Snitch with the GeoPerf's ground truth

to adapt to periodic latency changes. As time progresses, Java's CMA smoothing function becomes more resistant to change, until it cannot react to path order changes. As shown in Figure 4.5b when the Java and C++ drivers are compared under identical conditions, the Java driver demonstrates inferior performance in 80% of the cases. The problem identified here is related to CMA not being reset in the Java driver on a periodic basis.

Although these bugs might appear to be simple, they were not discovered previously. Common software testing often does not provide full test case coverage, thus it is hard to find bugs in application logic that affect performance, particularly in the presence of noisy input (e.g., network latencies). In the case of Cassandra, the official fix for the bug that we found also included a new test case to cover the specific scenario that we identified for the developers.

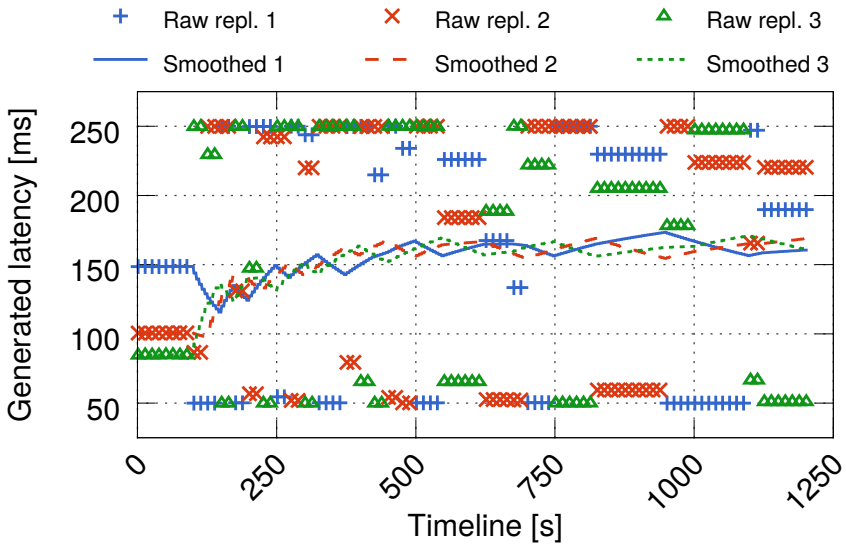
GeoPerf's running time We conduct our explorations using a cluster of 8 heterogeneous machines running Ubuntu 14.04, with a total number of 76 CPU cores and 2GB RAM per core. We use Cloud9 [52] to parallelize and distribute symbolic execution over these machines. Both sets of explorations finished in under 5 hours.

4.4.2 Evaluating the Impact of the Bugs

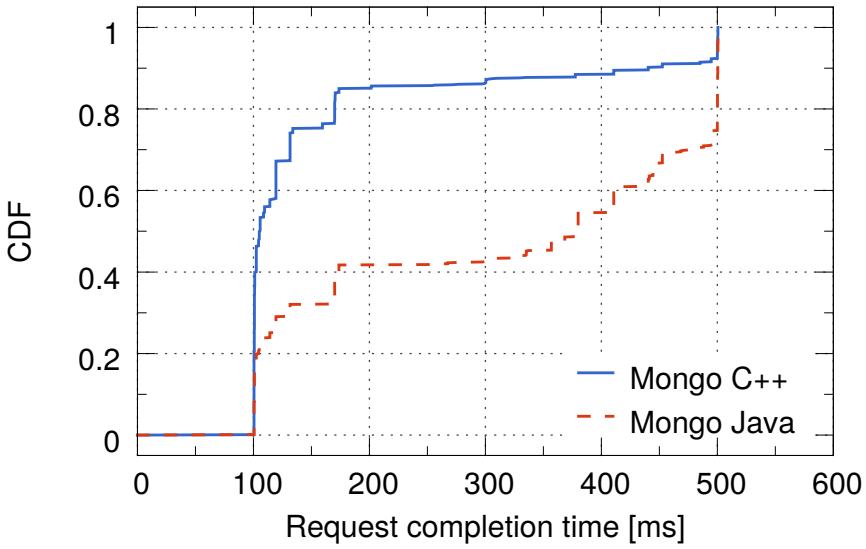
Here we quantify the potential impact of the bugs found by GeoPerf in both systems under real world conditions. We set up our simulations as in the previous section, and use 14 consecutive days of EC2 latencies (from Tue, 06 Jan 2015, 9GMT) obtained earlier as a concrete input set to GeoPerf. To consider the possible scenarios, first we group the latency samples based on the originating region. Then, from each group we pick combinations of triplets (as 3-way replication is a popular, straightforward choice). We sampled all 9 regions, where each region has 8 potential destinations, producing $9\binom{8}{3} = 504$ combinations in total. Figure 4.6 shows the excess median and 99th percentile time gained per request per day.

Each line corresponds to one of the 14 days of latency traces; each point on a line represents one of 504 possible deployments. For Cassandra's evaluation, instead of using the ground truth model we used the original version of the Dynamic Snitch and the fixed version of the same Snitch. Figure 4.6a shows that over 10% of all requests were affected by the bug. The median loss for 5% of all requests is above 50 ms**.

**We have identified an error in our previous evaluation [1] of the bugs' impact. The overall impact is lower than we initially estimated, however it is still significant; both bugs were fixed by the system's developers.



(a) Set of 3 path latencies generated by GeoPerf (points) and the smoothed version of these latencies(lines) after applying MongoDB Java driver's CMA



(b) Request completion time for MongoDB's C++ and Java drivers, as compared by GeoPerf

Figure 4.5: Comparing MongoDB's drivers using GeoPerf

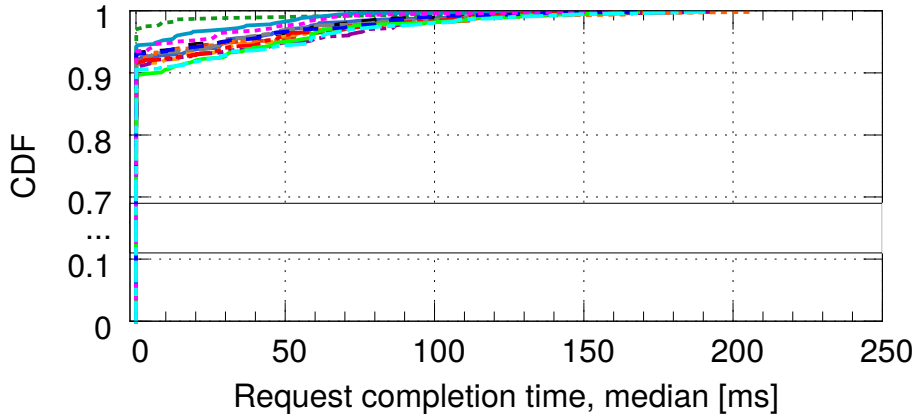
Next, we compare the C++ and Java MongoDB drivers. Figure 4.6b shows the effects of using CMA in a dynamic network environment on 99th percentile. Over 20% of all requests were unable to react to changing conditions, which resulted in a long tail. The negative CDF tail on both Figures 4.6a and 4.6b is explained by two factors. First, both systems choose a replica at random if it falls below a set latency threshold, which accounts for a certain amount of a slightly worse replica being chosen by the corrected algorithms. Second, when path latency shows a high variance there is a certain amount of inertia in both Cassandra's EDR and MongoDB C++ drivers EWMA, leading to additional when the actual latency has returned to its mean value.

In summary, these findings demonstrate the significance of the bugs found in both systems, and the potential unnecessary delays in cloud services due to these bugs.

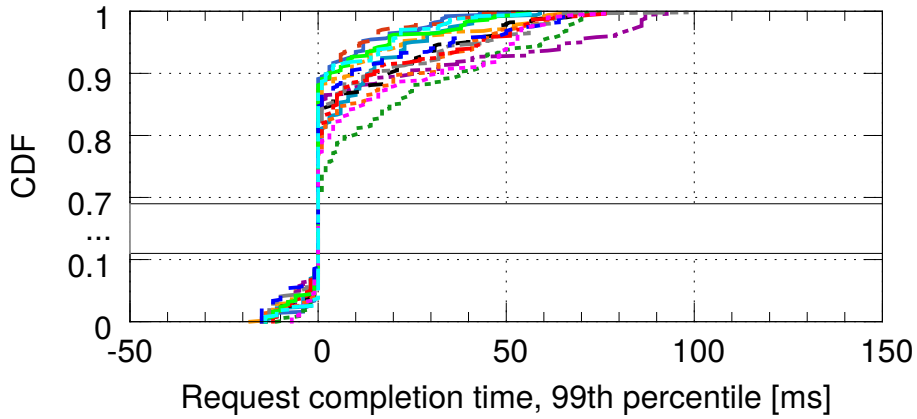
4.4.3 Reset Intervals

Cassandra's Dynamic Snitch flushes all sampling buffers every 10 minutes to allow temporary badly performing nodes to regain their score (see Section 4.2.1). Intuitively, such behavior appears to have negative consequences because the entire history of the smoothed data is lost. This results in all replicas being indistinguishable for a short period. To test this scenario we used GeoPerf to identify the effects of buffer resets within Cassandra's Dynamic Snitch.

To do so, we configured two simulations to use identical EDR smoothing functions and identical fixed versions of the Dynamic Snitch. However, in one of the simulations, we disabled periodic buffer resets allowing GeoPerf to generate a latency input such that these simulations produce different performance results. As GeoPerf utilizes iterative search (see Section 4.3.5) it attempts to find code paths that demonstrate performance differences within a limited number of latency inputs of a given iteration. Covering the initial time interval of 10 minutes of latency inputs for 3 nodes, would require hundreds of symbolic variables and hours of computation. Fortunately, the first 10 minutes of the simulation is not particularly interesting for us, because both algorithms would make identical choices (given their identical configuration). Therefore, we warmed up the simulation with concrete values from EC2 traces for the first 9.5 minutes. Next, we configured GeoPerf to initiate symbolic exploration from that state. To speed up exploration even further and simplify the test case, GeoPerf was configured to change latency only on a single latency path. Furthermore, we explored only two cases: when resetting sampling buffers has a positive and negative effect on performance.



(a) Cassandra's buggy version compared against fixed version



(b) MongoDB's C++ driver compared against Java drivers

Figure 4.6: The CDFs of the median and the 99th percentile request completion time difference of Cassandra and MongoDB respectively (EC2 latency trace replay via GeoPerf). Each figure contains 14 CDFs, one for each day of the trace of latency samples.

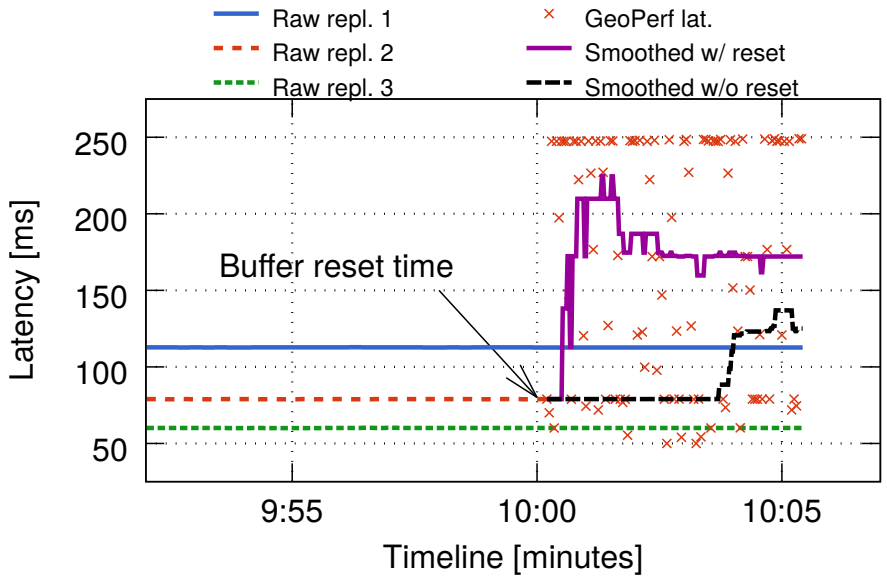
Positive effects of buffer resets Figure 4.7a demonstrates the positive effect and it shows two sets of latencies as observed by two simulations. The time interval up to 10 minutes represents the warm up period when both simulations were fed the same concrete latencies from EC2 traces. At 10 minutes the first simulation resets its buffers along all three paths. GeoPerf starts to generate latency inputs for the second (middle) latency paths indicated by the red dashed line. At this point, the middle path (red dashed line) splits into 2 paths. The purple line on top corresponds to the smoothed latency observed by the algorithm that resets its buffers, while the black dashed line, corresponds to the smoothed latency observed

by the second algorithm that did not reset its buffers. Finally, red points are the individual latency inputs generated by GeoPerf for the middle path (the actual latency). This figure demonstrates how the perception of the smoothed latencies differ by two algorithms. The buffer that does not have any samples reacts quicker to the latency change generated by GeoPerf, while it takes significantly longer for simulation with the full buffer to follow the change. The difference in perception of the network state affects the decisions of the replica selection algorithms. The first algorithm chooses optimal replicas *1* and *3*; while the second algorithm with the stale view selects paths *1* and *2*.

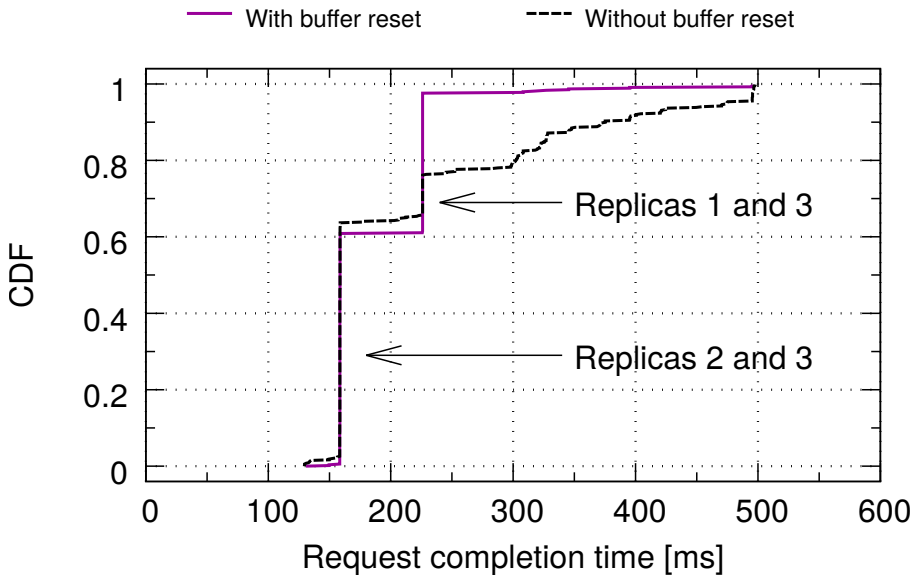
At 10:05 minutes GeoPerf's exploration has come to a halt. At this point, the buffer of the second algorithm has incorporated enough new samples to realize the change. Thus the perceived order of replicas again becomes identical for both simulations. It is unnecessary for both algorithms to have identical perception of the network state, as it is sufficient for replicas to be sorted in the same order. From now on, it is no longer possible to have divergent decisions by the two algorithms (until the next buffer reset).

Figure 4.7b is a CDF of request completion time of both algorithms over the tested time interval. The difference in the two graphs demonstrates the performance anomaly depicted in Figure 4.7a.

Because GeoPerf is based on the iterative heuristics that look for solutions that satisfy target performance ratios, not all latency samples picked by GeoPerf were strictly optimal. Certain latency values were below path 1 (blue line), making the choices of the second algorithm (without buffer reset) better than its counterpart. Setting a stricter search criteria for GeoPerf would partially solve this issue, at the cost of significantly increasing search time.



(a) Set of 3 path latencies generated by GeoPerf and the smoothed version of these latencies after EDR.



(b) CDF request completion time, the difference between Cassandra's Dynamic Snitches with and without buffer resets. The two arrows indicate request completion times realized by selecting two distinct sets of replicas.

Figure 4.7: GeoPerf found a case when resetting sampling buffers has a positive effect on performance.

Negative Effects of Buffer Resets To our surprise, GeoPerf was able to find a counterexample where resetting a buffer will degrade performance. We used the same configuration as in the scenario above, except in this case we swapped two algorithms by disabling periodic buffer resets of the first algorithm and enabling it for the second, thus causing GeoPerf to search for code paths that can demonstrate harmful effects of resetting the sampling buffer.

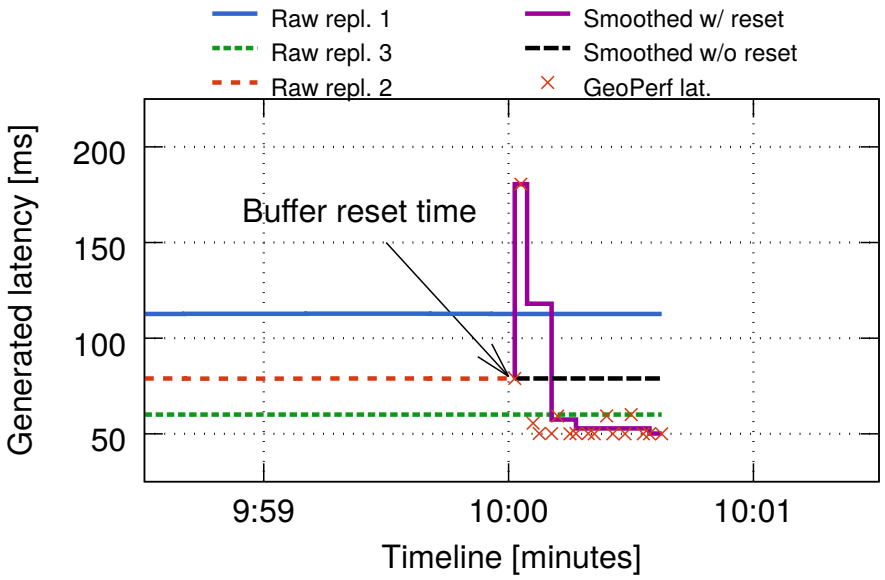
In Cassandra's Dynamic Snitch, latency sampling is temporarily decoupled from the replica decision logic (i.e., new latency samples are being inserted into buffers at run-time as requests are served), while buffer re-computation (e.g., re-evaluation) happens at discrete time intervals. GeoPerf was able to leverage this time difference between insertion of new latency samples into the buffer and the time of the replica decision, hence demonstrating a case when resetting a buffer might lead to a degradation in performance. When a buffer is empty, it becomes more susceptible to a change due to new latency samples. Therefore, a small number of samples can alter the state of the perceived smoothed latency. However, the reevaluation of the buffer happens at the discrete time intervals, which can result in a case where the perceived latency is wrong for the entire interval.

Figure 4.8a, demonstrates a sequence of three events: a buffer reset at 10 minutes is followed by a high latency spike on the path 2, which is followed by buffer reevaluation. This resulted in an incorrect perception of the smoothed latency for the duration of buffer re-computation intervals. In this example the empty buffer was too sensitive to a short term latency spike.

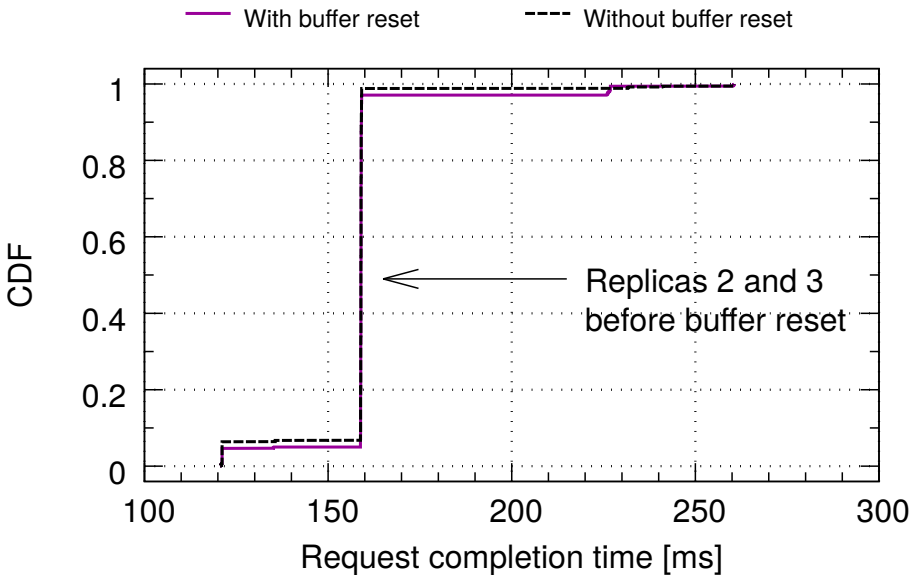
Figure 4.8b demonstrates the difference in the request completion time between these two cases. Due to the short duration of the incorrect perceived latency, this figure does not show a significant difference in performance. The duration of how long a replica selection algorithm can be wrong depends on the arrival rate of new samples. In the case of Dynamic Snitch this value depends on the rate of requests sent to a particular replica.

4.4.4 Evaluating Effects of Buffer Resets

In the previous section, we used GeoPerf to study implications of buffer resets as infrequent individual events. Next, we want to see if these individual events can have a cumulative impact on system performance and replica selection. To do this, we used our long term latency traces as concrete input to our simulations. Similar to Section 4.4.2, we used 14 days of consecutive latency traces and all possible geographic deployments on Amazon EC2 of 3 replicas. Utilizing real world traces we compared pairs of algorithms, where only one was configured to reset sampling buffers.



(a) Set of 3 path latencies generated by GeoPerf and the smoothed version of these latencies after FDS.



(b) CDF request completion time, the difference between Cassandra's Dynamic Snitches with and without buffer resets. The arrow indicates predominant request completion time by the two algorithms, before buffer reset.

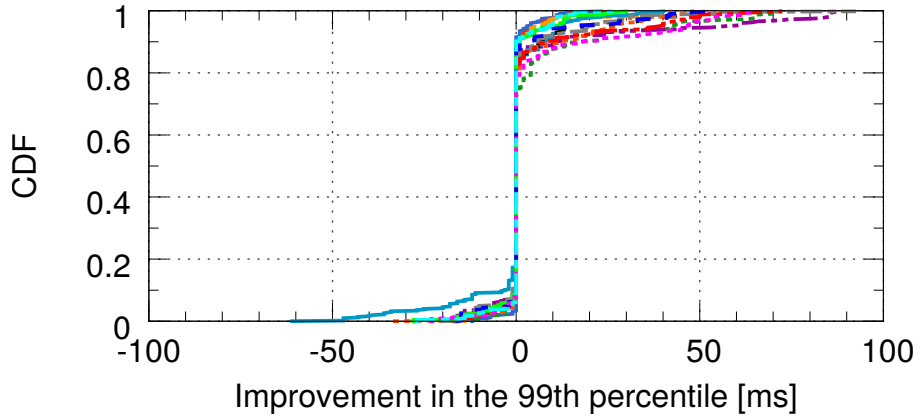
Figure 4.8: GeoPerf found a case when resetting sampling buffers can have a negative effect on performance.

Evaluating Effects of Buffer Resets on Cassandra's Snitch First, we evaluated effects of buffer resets on Cassandra's Snitch. As demonstrated above it is possible to encounter both cases when resetting a buffer will have a positive effect and a negative effect on performance. However, replaying real latency traces we did not notice any significant difference in the case of Cassandra. We increased reset intervals up to 1 second, and measured its effects on the 99th percentile; still observing no difference. Two factors can explain this result. **First**, the EDR algorithm used by Dynamic Snitch has a rapid adaptation rate, as samples within its buffer reservoir are skewed towards new arrivals. This allows the algorithm to react quickly to changes in latency trends. Therefore, if some latency samples are not representative of the current network state, EDR will quickly change its estimation and will not be incorrect for a long time. **Second**, due to decoupling of buffer resets and buffer reevaluation intervals, it is likely that the sampling buffer will contain a few samples by the time the next reevaluation is due. In such a scenario a buffer reset might have no effect on replica selection. Moreover, even the first sample that arrives in a buffer is representative of the current network conditions. Unlike an exponential moving average, where samples are multiplied by a given constant coefficient, having even a single sample in EDR is enough to correctly estimate the instantaneous network state.

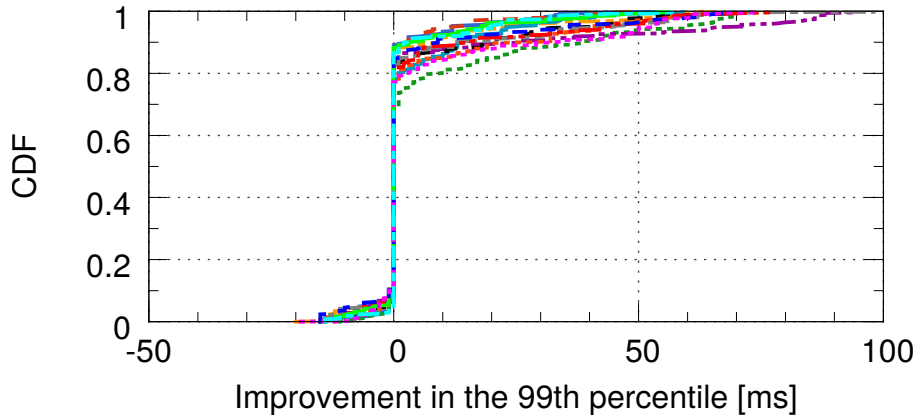
Evaluating Effects of Buffer Resets on MongoDB Next, we evaluated the effects of buffer resets on both MongoDB drivers: C++ and Java. The MongoDB Java driver is based on the cumulative moving average computed by Formula 4.1, where the cumulative sum of all samples is divided by the total number of samples.

$$CMA_n = \frac{S_1 + S_2 + ... + S_n}{N} \tag{4.1}$$

To estimate the effects of buffer resets and its sensitivity to the reset interval we evaluated two cases: when the reset interval was set to (i) 10 minutes and (ii) once every minute. Figure 4.9 demonstrates these two cases. As shown by GeoPerf this smoothing function cannot effectively track changes in network conditions (Section 4.4.1). The more samples introduced into the buffer, the less reactive the function becomes. Therefore, as demonstrated by Figure 4.9a, periodic buffer resets have a positive impact on the system's performance. In general, 20% of all deployments show improvement in the 99th percentile, while another 10% to 20% demonstrate a small reduction. The reason for this degradation in performance in certain deployments is related to the possibility of resetting a buffer just when current network conditions are about to change. For example, if latency is temporarily elevated, then we have a chance of being influenced by these samples and therefore, perform suboptimally for the upcoming 10 minutes. It is apparent that this might be worse than not making a change in a first place. However, the



(a) Buffer resets at 10 min intervals.

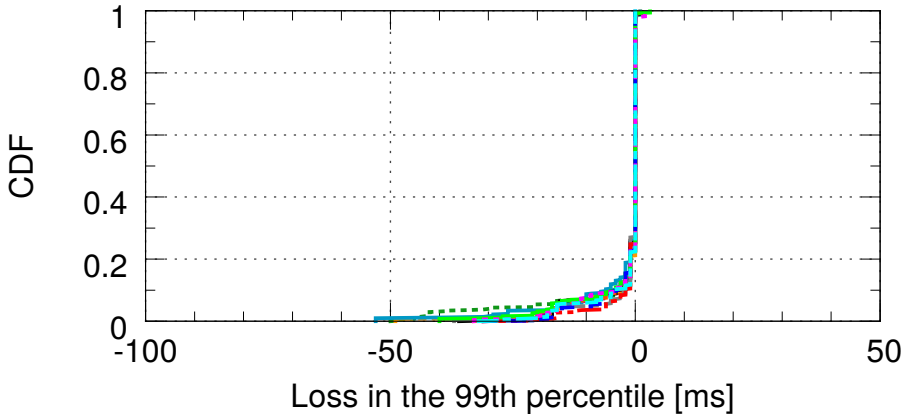


(b) Buffer resets at 1 min intervals.

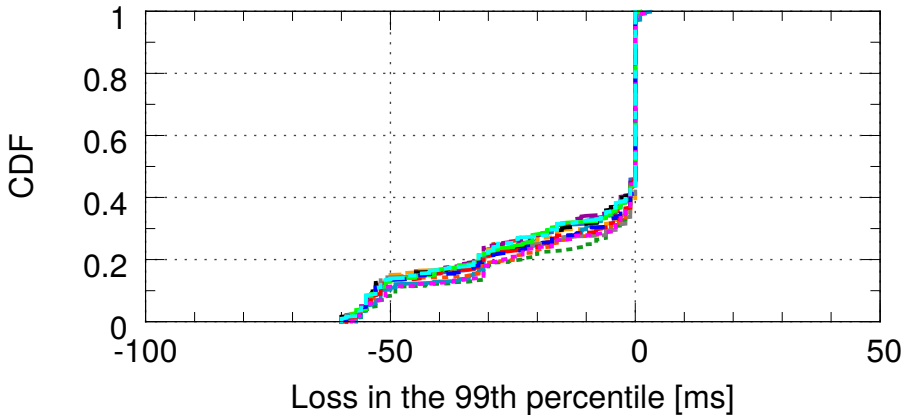
Figure 4.9: Effects of buffer resets on MongoDB Java driver with CMA.

adverse effect of such cases is less than the positive effect of buffer resets. We can also see that 60% of all deployments are unaffected because these deployments are relatively stable, and their mean latency is a good approximation the majority of the time.

To see the effects of the reset interval for the second case, we increased the buffer reset frequency to once per minute. Figure 4.9b illustrates the benefits of periodic resets on CMA. Similarly, 60% of all deployments were unaffected by this change. The reason for this is because long tail latency events that contribute to the 99th percentile are relatively infrequent. Therefore, to track them it is not enough to reset the buffer every minute.



(a) Buffer resets at 10 min intervals.



(b) Buffer resets at 1 min intervals.

Figure 4.10: Effects of buffer resets on MongoDB C++ driver with EWMA.

In contrast, the MongoDB C++ driver is based on an Exponential Moving Average with coefficient $\alpha = 0.25$, shown in Formula 4.2.

$$EWMA_n = \alpha \cdot S_n + (1 - \alpha) \cdot S_{n-1} \tag{4.2}$$

While this algorithm has a much faster adaptation rate than CMA as demonstrated by GeoPerf, it has a very significant limitation as EWMA requires a minimal initial set of samples to represent the actual network conditions. This occurs because all new samples are multiplied by the constant coefficient (in this case 0.25). Therefore, EWMA values of buffers with a small number of samples are misleading and unrepresentative of the actual network conditions.

These effects are shown in Figure 4.10 where we evaluated reset intervals of 10 and 1 minutes. Figures 4.10a and 4.10b demonstrate that EWMA suffers

significant penalties from periodic resets, and the higher the frequency of these resets the greater their negative effect on performance. Every time the EWMA buffer is cleared the replica selection algorithm loses track of the actual latency values. This can result in the algorithm making suboptimal choices. Both Figures 4.9 and 4.10 demonstrate that no single reset interval is optimal for all deployments (i.e., network conditions). As network conditions periodically change, latency smoothing functions would benefit from adapting their reset interval appropriately.

Discussion For dynamic systems to perform run-time decisions, it is important to have a correct representation of the network's state. Moreover, it is important to react quickly and timely to changes in this state.

Our evaluation studies of the effects of buffer resets on various smoothing functions used in modern replica selection algorithms demonstrate both positive and negative effects of these resets. The outcome often depends on the actual network conditions and frequency of changes, making static configuration suboptimal. This work demonstrates the dependency between sampling, buffer resets, and buffer reevaluation intervals. This raises important questions of when and how often to perform these operations. These questions are important as depending on the environment, even infrequent events can have a significant contribution to the overall system performance. As the result we argue that there is a need for an algorithm that can quickly recognize changes in network conditions and react appropriately, for example, by adjusting its buffer reevaluation interval or changing the sampling rate.

4.5 Limitations

This section outlines the list of current GeoPerf's limitations and potential future directions for improvement. First, before we can apply our method, we need to extract the replication selection module from an application in order to plug it into our simulation. If the logic itself is written in a programming language other than the target language processed by the symbolic execution engine, then this module needs to be translated to the target language in order to be executed. Our experience with two different systems implemented in two different programming languages shows that the translation effort is minor.

Next, despite our performance optimizations, state explosion is still a real problem and potentially slows down the search process leading to the search becoming impractical. This might become an important factor if additional parameters will be incorporated into the tested models (e.g., system load, queue sizes, etc.).

Another limitation is related to the identification of the actual underlying

problem in a given replica selection algorithm. Symbolic execution combined with the parallel simulations can show the presence of a problem in one algorithm and provide a reproducible trace of the events that lead toward this difference in performance, but this method does not explicitly identify the line of code that leads to this difference. Having an identification at the level of the program statement within the algorithm would make it much easier to perform backtracking and understand the cause for the difference in behaviors.

Finally, our heuristics represent our best current effort to find divergence between the two algorithms being compared, but do not guarantee a full coverage. Despite this limitation we have been able to spot problems in the replica selection algorithms used by the two systems that we have considered. This method does not reveal the maximum performance difference that can exist between two algorithms, but rather we find an average difference over a period of exploration.

4.6 Summary

In this chapter, we first demonstrate the need for dynamic replica selection within a geo-distributed environment on a public cloud. Second, we propose a novel technique of combining symbolic execution with lightweight modeling to generate a sequential set of latency inputs that can demonstrate weaknesses in replica selection algorithms.

We have created a general purpose system capable of finding bugs and weaknesses in replica selection algorithms. We believe this is the first tool for systematically testing geo-distributed replica selection algorithms. We use our system GeoPerf to analyze the replica selection logic in Cassandra and MongoDB datastores, and find a bug in each of them.

Chapter 5

Improving Network State Estimation using EdgeVar

In the previous chapter, we demonstrated a novel technique of performing systematic testing of replica selection algorithms. The aim of this technique is to validate and improve the core logic utilized by geo-distributed applications in making run-time decisions. However, while correctness of these algorithms is paramount, the input data that is being fed into these algorithms (i.e., the perceived network and system state) has a profound impact on the quality of final choices. For example, for a geo-distributed system, an incorrectly estimated network latency, fed into a "perfect" replica selection algorithm can result in a suboptimal decisions, leading to consecutive performance degradation of that system. In this chapter, we look into improving existing latency estimation techniques in order to provide a better perception of network state for geo-distributed systems.

Using our traceroute measurements across Amazon EC2, we identified distinct latency classes and correlated them with changes in network paths. We demonstrate that these classes persist over the period of our measurement and can be used as indicators for persistent change in network latency. Based on these observations, we designed and implemented EdgeVar, a system capable of distinguishing between latency changes due to routing changes and congestion build up. Using EdgeVar we removed the effects of routing changes from the latency stream, thus providing a clear view into network latency. Using this technique, we improve the quality of latency estimation, as well as the stability of network path selection.

The rest of this chapter is organized as follows. In Section 5.1 we introduce the problem and show real world latency traces that motivate our work. In Section 5.2 we outline the architecture and the main components of EdgeVar. In Section 5.3 we discuss One Way Delay (OWD) measurements and correlate network path

changes to individual latency classes. Next, in Section 5.4 we introduce Front Runner (FR) algorithm and discuss the details of identifying changes in latency levels. We evaluate EdgeVar in Section 5.5. Finally, in Sections 5.6 and 5.7 we discuss limitations and conclude this work.

5.1 Introduction

To deliver the best possible service to clients, geo-replicated systems need to adapt to ever-changing conditions by performing real-time network sampling in order to estimate selected characteristics of the network's state. The selected metrics of interest often include the mean, median, and 99th percentile of the network latency. Subsequently, these estimates are used to make run-time decisions, such as replica selection, placement, and data dissemination. Common techniques for sampling network latency include variants of moving average and reservoir based sampling. For example, the majority of MongoDB [7] drivers rely on exponential weighted moving average (EWMA) when selecting the nearest server, while Cassandra's [6] replica selection algorithm uses a biased reservoir sampling technique [46] to select a subset of the best replicas.

These reservoir sampling techniques perceive a very narrow view of the network's *true* condition, often discarding the important properties of the very metric that they attempt to estimate. For example, a typical reservoir based approach only estimates a single percentile, while ignoring the properties of the sampled distribution preserved in the reservoir. By treating network latency samples simply as a data stream, these techniques are oblivious to the underlying nature of network latency. The latency of a network path is a composition of routing delay (composed of the propagation time over the physical distance along the media between the source and destination plus receiving and forwarding delay at each router) and the amount of competing traffic on that path (which results in network queuing and congestion). Currently, these individual contributions cannot be differentiated using existing techniques, and this knowledge cannot be exploited. This can lead to suboptimal decisions of higher application logic (such as replica selection) which would benefit from an accurate perception of the network.

In this work, we propose a novel technique for network latency estimation, which combines knowledge of the underlying infrastructure of the WANs (currently employed by cloud providers) to clearly differentiate the delay associated with network routing from the delay due to network queuing and congestion. For geo-distributed systems, this provides a clear view of the network and allows applications to make better choices.

We performed detailed network measurements across geographic regions of Amazon EC2, a popular cloud provider. We combined traceroutes with OWD measurements in order to correlate observed network paths and path classes with

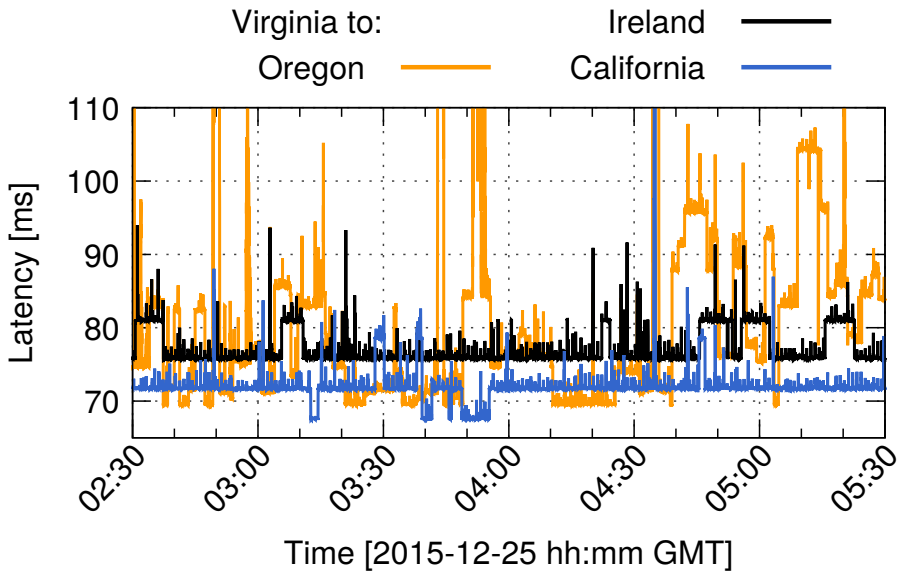
distinct latency classes, i.e., matching routing changes to changes in network delay. Using this analysis, we were able to isolate individual network delay contributions coming from changes in routing and those due to network queuing and congestion. Using properties derived from our measurement, we designed and developed an algorithm that can perform this separation at run-time based on common (already used/in place) latency sampling. The output of the algorithm is the base network latency (i.e., zero frequency component) and the residual latency variance. The precision of this algorithm can be further enhanced by periodic, timed network path tracing and prior knowledge of the network path, when the algorithm operates. The complete list of EdgeVar's contributions is listed in Section 1.3.

Figure 5.1a shows RTT measurements from the Amazon EC2 datacenter in Virginia to three other geographically distributed datacenters. Each network path demonstrates several latency levels that periodically reoccur during the trace; latency levels on different network paths change at different moments in time. Excluding shifts in the latency level, each network path demonstrates low latency variance. In Section 5.3 we show that these changes in latency levels are correlated with routing changes on these network paths.

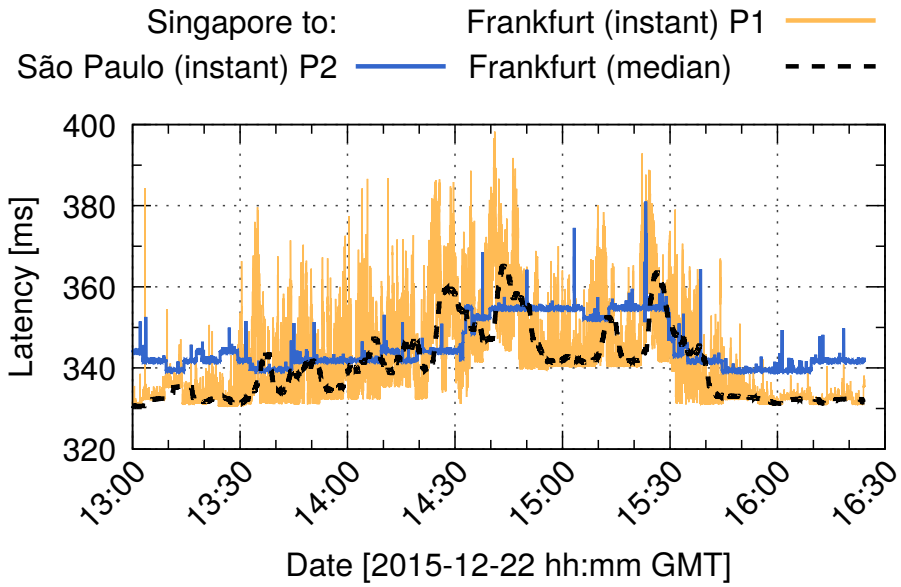
Figure 5.1b shows two network paths with high and low latency variances (P1 and P2 respectively). For a tool that relies on the single metric of the median or rolling average, it is difficult to consistently differentiate these two qualitatively distinct network paths. For example, the dashed line in Figure 5.1b shows the median latency of P1 computed over 5 minutes*. The median value lags behind real changes, causing alternating selection between these two network paths. Selecting P1 for any period in the interval between 13:30 and 15:45 will lead to a longer latency tail. Reducing the reservoir size, thus increasing the sensitivity of the median estimation, will *not* solve the problem, but rather increase the frequency of flapping between these two paths.

In this situation one solution is to "understand" the base level latencies and the latency variance of each network path and exploit this to make an informed decision while taking into consideration the median and tail Service Level Objective (SLO) of a running service. For example, an application may prefer to switch persistently to using the lower variance but higher base latency path (P2) for the duration of the congestion on the alternative path (P1). In this paper, we seek to provide a better view of the network's conditions by separating the persistent network delay associated with the route from the highly variable residual component due to network queuing and congestion.

*The median RTT of P2 (Singapore to São Paulo) closely follows the actual latency graph and thus, is not shown for visibility purposes.



(a) RTT measurements from datacenter in Virginia to three other geo-distributed datacenters. Each network path demonstrates several delay levels. Changes in the delay level occur at different times on different network paths.



(b) RTT measurements from datacenter in Singapore to Frankfurt and São Paulo. It is necessary to have precise knowledge of the delay variance and latency levels to make an informed decision during path selection.

Figure 5.1: Two sets of RTT latency traces.

5.2 Architecture

EdgeVar is our network measurement and classification tool and it is designed to be integrated into existing geo-distributed systems that depend on the network's conditions to make run-time decisions. The purpose of this tool is to provide geo-distributed systems with a clear and more reliable perception of the network's state to act upon. EdgeVar performs run-time network path classification and builds up a set of corresponding latency classes. Using this knowledge, it breaks up a noisy latency signal into components of persistent routing delay and residual latency due to queuing or congestion buildup. EdgeVar is implemented as a library, positioned between the network measurement infrastructure and higher application logic that relies on network latency. In Section 5.5 we show how EdgeVar can be used to estimate network latency and perform network path selection.

Figure 5.2 illustrates the main components of EdgeVar and two possible execution models. EdgeVar can operate on latency sampling alone (left box below step #1), or it can benefit from online or offline traceroutes (right blue box below step #5). The primary input into the system is network latency measurements. In Step #1, an application's existing infrastructure is used to provide EdgeVar with latency samples coming from relevant network paths (paths of interest). For each measured path, EdgeVar maintains a finite size reservoir containing raw latency samples (step #2).

In Step #3 we apply FR (Section 5.4.3), our domain specific, online implementation of a step detection algorithm [53, 54]. Using statistical analysis FR identifies the time and magnitude of latency steps, i.e., shifts. This identification allows us to extract a base level delay associated with a given network path between datacenters. We subtract this base level from the raw latency samples (obtained in step #2) to produce the residual latency. Working with this residual latency signal allows a better estimate of the distribution of the samples: their variance and "tailedness". These better estimates are important for many tail sensitive applications [6, 7]. EdgeVar's output is presented to higher level application logic (step #4) that makes a decision based on an explicit representation of the network's current condition.

Using FR to identify the latency class (characterized by a specific base latency of the latency step) introduces a short delay from the time of when the latency class has changed to the moment when FR reports it. This happens because FR requires a few samples to distinguish persistent change from a short-lived latency spike. Data from traceroutes can be used to minimize this delay (step #6).

In Section 5.3.2 we demonstrate that latency classes and their magnitude can also be obtained by performing periodic traceroutes along a network path. The presence of a particular IP address along the network path, from a source to a destination, can indicate a distinct latency class. By performing periodic

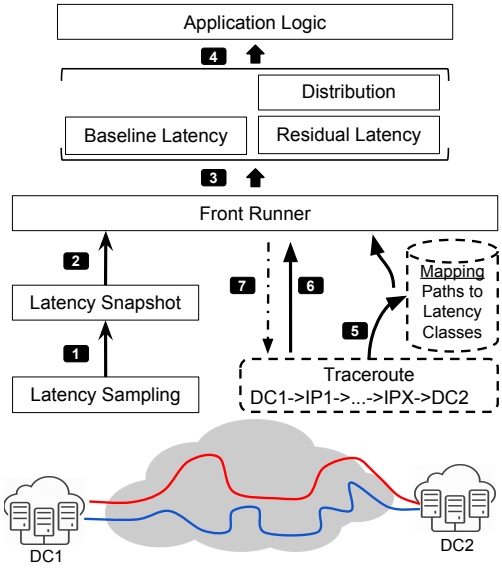


Figure 5.2: EdgeVar's architecture

traceroutes we can build up a mapping between path classes and latency classes (step #5). This allows traceroute to identify the expected base latency of each observed network path (Section 5.3.2). Finally, in Step #7, FR can initiate a traceroute when it suspects there is a change in a latency level, thus further reducing the latency class identification delay.

5.3 Correlating Network Paths with Latency

In this section, we describe the network latency and traceroute measurements performed over 10 geographic regions of Amazon EC2. Using our measurements we demonstrate a correlation between network paths and network latency. We show that a given network path produces the same network delay, i.e., packets traveling a previously observed path incur the same network latency as previously observed, excluding additional delay due to potential network queuing and congestion. We account for congestion in Section 5.4.

By deploying virtual machines (VMs) across 10 geo-distributed datacenters[†] we performed full mesh experiments collecting path traces and latency samples over 2 weeks. We utilized Paris traceroute[55] to trace the paths taken between each pair of datacenters twice per minute. We synchronized our measurements using the local Network Time Protocol (NTP) [56] service. Each pair of geo-distributed VMs initiated tracing between each other at the same time.

[†]Currently available EC2 datacenters are in Virginia, California, Oregon, São Paulo, Ireland, Frankfurt, Singapore, Tokyo, Sydney, and Seoul.

To trace network paths, Paris traceroute iteratively sends three probes[‡] towards the destination IP, each time incrementing the probes' Time To Live (TTL) value, starting with one. When probes are forwarded by network elements, their individual TTL values are decremented by 1 at each hop. Finally, when the TTL value is zero, the last encountered router sends an ICMP Time Exceeded notification to the message originator. This notification typically contains the IP address of the last router. Hence, by collecting these notifications, Paris traceroute reconstructs the entire network path to the destination; moreover, by measuring the difference in local time between the probe being sent and the ICMP message being received, the program estimates the RTT to each hop of the network, including the destination. Unfortunately, not all routers reply with an ICMP response when TTL reaches zero. This leads to some network paths being only partially uncovered. To address this issue we considered three scenarios; we first treat each unknown IP address in a network path as unique, thus each path containing at least one unknown hop is a unique path. In the second scenario, we considered two network paths to be equivalent, if they have the same number of hops and all known IP addresses match at corresponding hops, i.e., a more relaxed condition. Such paths create path classes. Finally, we considered a case when an unknown IP address would match any known IP address at the corresponding position. We found that all three types of grouping match the observations described in this section; however, truly unique, fully identified paths are few in numbers (or require additional grouping), hence, they are not shown. In the rest of this section we used the second type of network path grouping.

In comparison to the original traceroute [57], Paris traceroute provides better precision in identifying network paths. In particular, Paris traceroute (from here on simply traceroute) is better suited for networks utilizing load balancing. This improved precision is achieved by maintaining flow affinity of traceroute's probes via smart control of the packet header. The tool keeps constant the 5 tuples (protocol, source and destination, IP address, and source and destination port numbers) that identify the flow and the subset of the packet header that is used during load balancing, as identified by the authors of Paris traceroute. This guarantees that in the presence of per-flow and per-destination load balancing, all probes of a single trace will likely travel the same network path towards the destination, providing a consistent view of the network.

However, the problem of the per-packet load balancing remains as network elements do not keep the packets of the same flow together, instead, their only concern is load balancing; thus, a packet's header is not considered when making a forwarding decision. Due to the random nature of such balancing, it is impossible to guarantee that the path recorded by traceroute is the path traversed by the last

[‡]We used UDP probes.

(three) probes that reached the destination. While per-packet load balancing of some routers [58] uses simple round-robin techniques to choose an output path for packets, we can not guarantee that this is the general case. In our measurements we did not encounter artifacts that could be explained by per-packet load balancing across network paths with a different delay. Therefore, we are inclined to believe that if per-packet load balancing were used along network paths where we performed traceroutes, it had no significantly affect on our measurements.

We seek to measure and map network latency to the corresponding network paths. By monitoring network delay on a single path, we reason that a path always produces a constant network delay (excluding variance due to congestion and network queuing delay and infrastructure upgrades). Having establish this relationship we can exploit this knowledge of the current path to reason about the network's conditions, such as expected network latency and variance.

5.3.1 Measuring One-Way Delay

Unfortunately, using traceroute alone does not prove a complete picture of the network's state. For each instance of a trace, we learn the network path from the source VM to the destination VM; however, the measured RTT also includes the probes' return time, which we do not know. Hence, the RTT observed on a single network path can change due to changes in the reverse path taken by traceroute's probes. To address this limitation, we measure one-way network delays and correlate these delays with the forward network paths identified by traceroute. To do this we, **first**, modified Paris traceroute to log timestamps when each probe is sent. **Second**, at the destination VM, we deployed a listener application that monitors the arrival of the traceroute probes and logs local timestamps upon their arrival. **Finally**, using NTP, we synchronized the set of 10 geo-distributed VMs, in order to accurately match time stamps among the datacenters.

Figure 5.3 demonstrates 24 hours of traceroute measurements between VMs deployed in Ireland and Oregon. This figure contains 5 plots. The top two graphs show the RTT latency observed by the traceroute's probes sent from each datacenter. The three graphs at the bottom show the corresponding one-way latency measurements obtained by using our modified version of the Paris traceroute (yellow and blue graphs) and the average one-way latency (the middle graph in purple).

As can be seen in Figure 5.3, the one-way latency measurements demonstrate negative correlation and increased variance. This artifact is caused by the local clock drift at each VM, as the clock in one of the two VMs runs faster than the other clock causing an incremental increase in the relative time difference (Δt) between the two VMs. This inflates latency measurement by Δt in one direction,

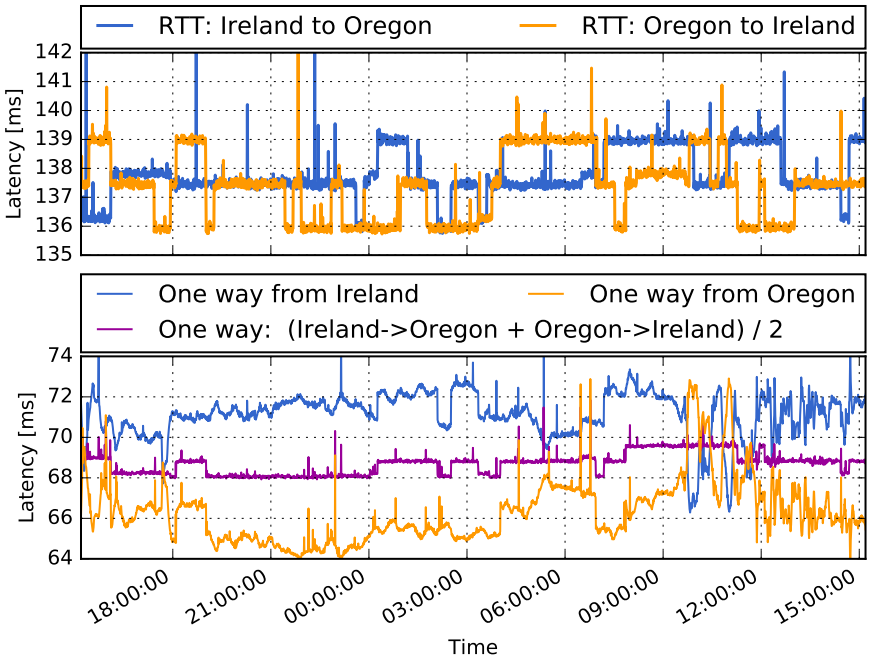


Figure 5.3: The top graph shows two sets of RTT measurements initiated from Ireland and Oregon respectively. RTTs are different because probes took different network paths across the network. The bottom graph shows OWDs measured from each of the two datacenters and the average OWD (purple line in the middle) computed to remove clock drift among VMs.

while simultaneously decreasing latency measurements in the opposite direction. By computing the average one-way latency we eliminate this Δt component, this gives us half of the delay that a packet would encounter during a round trip between these VMs if it follows the same paths reported by traceroutes from each location. In other words, we have a pair of network paths and the total RTT latency of this pair. While the NTP synchronization between two VMs is not always precise, we can rely on the fact that the clock drift between two machines in the interval corresponding to the RTT is insignificant for our purposes.

Figure 5.4, shows a close-up view of a portion of the curves shown in Figure 5.3. Note, that every change in the latency level of the average one-way latency corresponds to a co-directional change of double this magnitude in one of the two RTT measurements in the top graphs; however, the opposite does not hold, because the RTT measured from either datacenter might be affected by the return path of the traceroute probes, which is not observed in this measurement. For example, at 7:10 we can identify a rise in the average one-way latency which corresponds to the latency jump in the one-way latency observed from Ireland to Oregon and similarly, to the increase in the RTT measurement between these

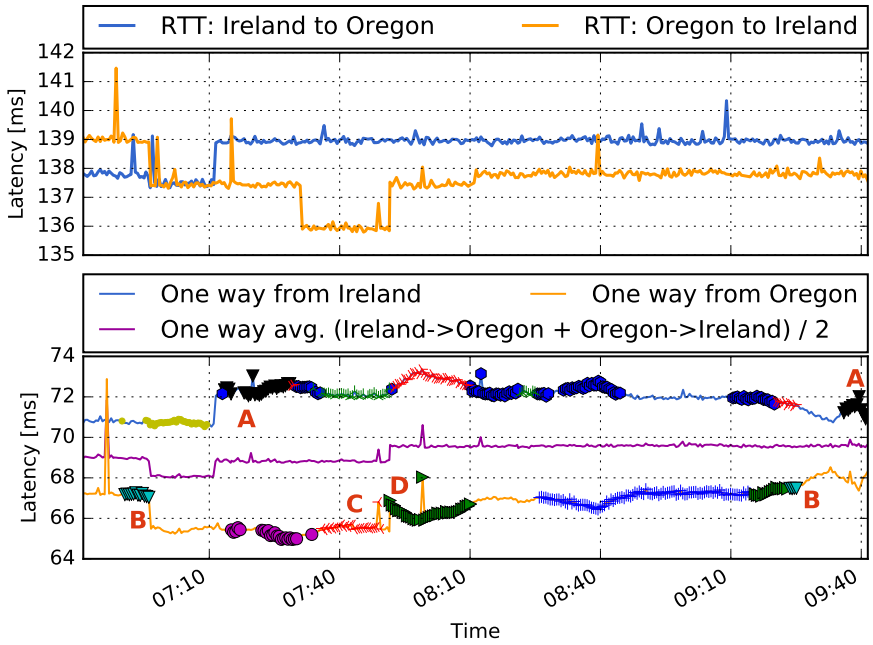


Figure 5.4: The 3 hour close up view of Figure 5.3. The bottom plot shows the sets of markers placed on the OWD graphs as measured from each datacenter. The markers highlight the most frequently used network path classes during the period of this trace. Labels highlight multiple occurrences of network paths classes. Certain changes in network path classes correlated with shifts in latency classes.

datacenters. However, a drop in the RTT from Oregon to Ireland at 7:30 is not reflected in any of the one-way measurements.

Figure 5.4 contains markers which indicate five of the most popular network path classes observed in each direction over that period. The empty spaces on each line correspond to less frequently used network paths and these have not been shown for the visibility purposes. **First**, we can see that the majority of network paths are utilized continuously over unevenly distributed time intervals. **Second**, closer investigation shows, that despite visual distortion caused by the clock drift, each path class corresponds to a single latency class (level). For example, the path classes labeled **A** and **B** in Figure 5.4 appear at different times in the corresponding traces; however, each time they have the same latency level as previously. Similarly, the transition from path class **C** to **D** in the Oregon to Ireland trace, correlates with a latency change.

The clock drift between the machines makes it hard to correlate network paths classes with the latency levels. Therefore, it is difficult to quantify these results

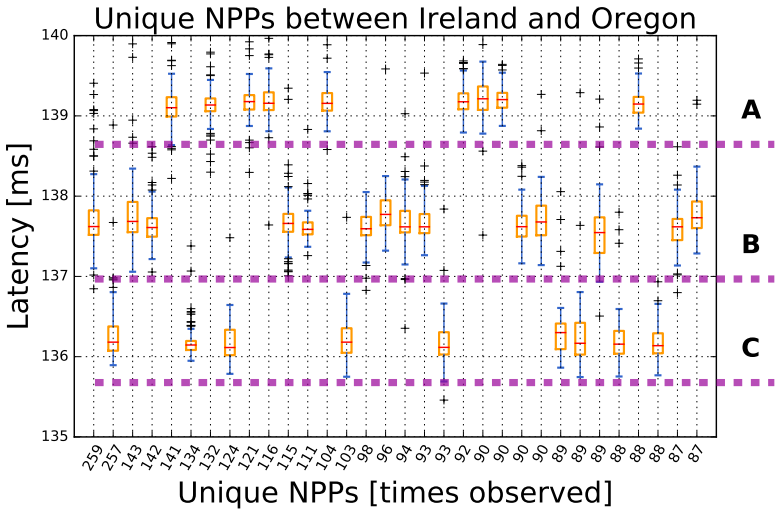


Figure 5.5: Latency distribution of the most frequently observed NPPs between Oregon and Ireland. The bodies of the boxplots indicate the median, first, and third quantiles, whiskers extended for 1.5 interquartile range (IQR) in each direction, the points lying beyond that point are indicated as crosses. Min. latencies are shown as dashed horizontal lines.

and draw a conclusion about all geo-distributed paths of Amazon's EC2 network. Unfortunately, using NTP, we were unable to synchronize clocks to a finer precision to reduce one-way latency distortion due to clock drift. In the next section, we demonstrate a correlation between network paths and latency using the average one-way latency.

5.3.2 Method

In this section, we demonstrate a correlation between distinct network paths and one-way network latency. We use the observation from the previous section, that the sum of the average one-way latency in each direction gives us the precise value of the RTT delay, between two geo-distributed datacenters, in keeping with the exact network paths traversed by the traceroute's probes. Therefore, showing the correlation between individual network paths by demonstrating the correlation of Network Path Pairs (NPPs) with RTT latency.

Figure 5.5 shows the latency of the most frequently observed NPPs between datacenters in Oregon and Ireland during our trace[§]. Boxplots indicate the latency distribution observed over all of the measurement period. Each individual latency

[§]Corresponding RTT measurements previously shown in Figures 5.3 and 5.4

sample is a combination of a network path's base latency and the residual latency caused by competing traffic. The latter component is variable and dependent on the network's conditions at the time of a measurement, thus we use the minimum observed latency as a distinct classification characteristic for each network pathP. This figure demonstrates three distinct latency classes formed around 138.7, 137, and 135.7 ms, labeled **A**, **B**, and **C** respectively.

Note the small standard deviation (small variance) of samples in all NPPs; from this observation, we draw the following conclusions. First, packets traveling over an network pathP observed earlier will encounter the same (comparable) latency. Next, the amount of the deviation is comparable to the latency variance at the time of the measurement (see Figure 5.3). Finally, the similarity in the amount of latency variance, indicates that changes in network paths are *de-correlated* with the changes in network latency variance.

On average there are 16 hops between the datacenters in Oregon and Ireland in each direction. The NPPs within each individual latency class (**A**, **B**, or **C** in Figure 5.5), differ by a small set of IP addresses at specific hop positions from other NPPs of the same class; hence these IP addresses can be used to further classify these NPPs into a distinct class. Similarly, NPPs that belong to a different latency classes (such as **A** and **B**) have unique, differentiable hop IP addresses. Therefore, the presence of a particular IP address at a specific hop position can be used to differentiate among NPPs belonging to different latency classes. The set of IP addresses is distinct for each latency level. Such differentiation could be used to give an early prediction of a change in routing. The presence of a particular IP address at the N-th hop from the source, along the path to the destination, can indicate a *persistent* change in routing (and therefore identify a specific latency class) *before* standard latency probes reach the destination and return (given that N is less than the length of the complete network paths).

Separating One-Way Latencies. We extend our method of correlating unique NPPs to latency to identify effects of routing changes on one-way latency. While we cannot identify one-way latency in absolute terms, we can isolate the effects of routing changes on one-way latency. For example, a pair of forward and return network paths (A, B) has changed between two consecutive traceroutes to a partially new pair (A, C), i.e., only the return network path has changed. By knowing the average OWD of each measurement and assuming a sub-millisecond clock drift in between two consecutive traces, we can infer the effect of a unidirectional path change by simply subtracting the first measurement from the second. Therefore, we can determine the absolute delay latency difference between network paths B and C.

Next, we used our traces to validate this property. Unfortunately, due to a large number of unique paths in each direction, in one week of measurements

we observed a very small number of cases when two paths repeatedly change to produce a different latency level. The majority of observed cases indicate a sub-millisecond latency change. While we were able to identify some cases that strongly correlated with the routing changes we were not able to draw conclusions about the entire network. However, we are inclined to believe, that with an extended period of sampling this property will hold across all network paths of Amazon EC2.

Geo-distributed paths across Amazon's network differ in the number of unique paths observed among each pair of datacenters. Over the two weeks of measurement, the majority of datacenter pairs demonstrated a decrease in frequency of encountering new network paths. Moreover, the popularity of network paths is non-uniformly distributed; certain network paths are used very frequently while others occurred only once. Without considering infrastructure changes and new peering agreements, our observation suggests that there is a finite (and small) number of possible paths between any two datacenters in EC2. Thus, the most popular and influential network paths can be identified and used for latency decomposition in an acceptable time frame.

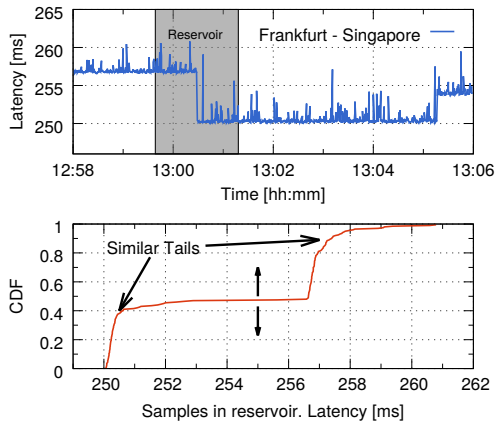
There are many other properties that can be computed using this data, hence we are making this data publicly available in the hope that other researchers will use it to find interesting statistical properties. In this section, we demonstrated a strong correlation between network path classes and network latency. In the following section we show how network path classes can be inferred from the latency trace.

5.4 Detecting Changes in Latency

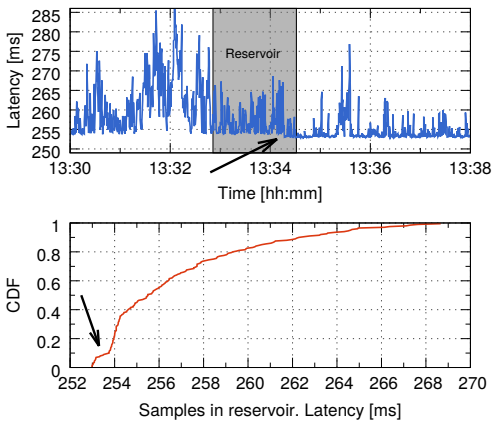
In Section 5.4.1 we study the possibility of detecting changes in network delay classes by monitoring the distribution of samples within the reservoir under different network conditions. Then, in Section 5.4.2 we outline desirable properties of an algorithm that could detect changes in delay classes at a run-time. Then, in Section 5.4.3 we define *Front Runner*, an algorithm for monitoring changes in latency classes and demonstrate its use.

5.4.1 Analyzing Temporal Changes in Reservoir Distribution

In this section, we study temporal changes in the distribution of samples within the reservoir when network conditions change. We use two distinct latency traces sampled from one network path during different time periods: the first trace, shown in Figure 5.6a contains instantaneous shifts among latency classes and the



(a) Latency trace (above) and the corresponding reservoir state (below), at the time of a sudden change in the latency level.



(b) Latency trace (above) and the reservoir state (below) during network congestion.

Figure 5.6: The top pair of graphs demonstrates the latency traces, two vertical lines, in each graph, show latency samples included in the reservoir windows (200 samples). The left red vertical line is the start of the samples in the reservoir, while the right green line is the sample just being added to the reservoir. The bottom pair of graphs shows the CDFs of latency samples in the reservoirs.

second trace depicted in Figure 5.6b demonstrates the case of network congestion. The CDF plots in the lower part of each figure demonstrate the states of the corresponding reservoirs showing distributions of latency samples of each trace. In both cases we study the states of the reservoir before, during, and after a highlighted latency change[¶].

5.4.1.1 Discrete Latency Change

Figure 5.6a shows an interval of the latency trace and corresponding state of the reservoir when a change in latency class occurs. In Figure 5.6a, at 13:00 the latency dropped from 257 ms to 250 ms. We view this as a change in the latency class (based upon clustering different latencies into classes). The selected moment in time demonstrates the change in latency classes roughly in the middle of the reservoir; thus, half of the reservoir's samples belong to the 257 ms latency class, while the rest belong to the 250 ms class. The separation between these two sets of samples is aligned with roughly the 50th percentile and appears as a horizontal line in the CDF plot. As time progresses, the reservoir window slides right along

[¶]Several video clips demonstrating changes in real-time, are at: https://goo.gl/InDchH

the latency trace (the X axis), as it does so - this horizontal separation line slides up the Y axis in the CDF plot. Conversely, when a low latency class changes to a higher latency class, the horizontal line moves down the Y axis of the CDF. In Figure 5.6a, this is highlighted by the two vertical arrows drawn at roughly 255 ms.

Our first observation is that this case corresponds to the worst case scenario for detecting a change using the median of a reservoir. In this example it will take close to half of the reservoir's number of samples to change the value of the median latency, thus the median is a bad indicator of shifts in latency classes, hence alternative techniques are needed.

Our second observation focuses on the similarity between the tails of the new and old samples within the reservoir (highlighted by the two arrows marked "Similar tails"). The short tail near the 95th percentile corresponds to the latency spikes of the old samples, while a small latency dip around the 40th percentile is the future tail of the new samples, i.e., samples from the new 250 ms latency class. The similarity of these tails suggests that the change in latency classes (potentially caused by a routing change) does *not* produce a change in latency *variance*.

5.4.1.2 Latency Change due to Network Congestion

Figure 5.6b shows the state of the reservoir during a period of network congestion. In contrast to the first case, congestion increases (or decreases) over time, this produces a slower effect on the distribution of samples in the reservoir. Note that the reservoir's CDFs of the two different latency traces are structurally distinct. In Figure 5.6b we observe a curved CDF, where all samples above the 60th percentile are affected and tilted (spread) towards the tail. In contrast, in Figure 5.6a the latency samples fall along almost vertical lines in the CDF.

Our first observation is that despite the uneven distribution of samples in the reservoir, it is possible to identify the change in latency class. The black arrows in the trace and CDF plot both indicate a small change in latency classes. Similarly to the case shown in Figure 5.6a, as time progresses, the reservoir window slides further right, leading to the distinct artifact sliding up the axis of the CDF; however, soon there after, it becomes less evident. This occurs due to the new, high variance, latency samples, being inserted in to the reservoir. This suggests that during network congestion it is easier to identify routing changes at earlier stages; as more samples may obscure a change in latency class, hence delaying identification of this change in class. We conclude that *the majority of samples in a reservoir may be unnecessary for realizing changes in latency classes.*

5.4.2 Detecting Run-Time Shifts in Latency Classes

In this section we discuss the requirements of an algorithm for detecting changes in latency classes. Unfortunately, the commonly used technique of tracking the median percentile is not efficient for this task. As we have seen earlier it takes too many samples to realize that there has been a change. From Section 5.4.1 we know, that the majority of samples within the reservoir are irrelevant for detecting a change in latency class. Moreover, in the case of a high variance (see Figure 5.6b), the longer we wait the harder it is to distinguish the change. Fortunately, this property aligns with our desire to identify a change as soon as possible, without the need to completely fill the reservoir with new samples.

A latency trace is a lossy representation of a network's state. When observing the current delay we must use our judgment to decide upon the current latency class; but in some cases, the latency trace alone is insufficient. For example, a period of network congestion resulting in a high network variance which "hides" a group of lower delay classes was shown in Figure 5.6b. Therefore, the first question that we want to answer is how to clearly identify a change. This is followed by the question of how to make this identification robust and fast.

Physical properties of each packet's propagation across a network, suggest that the minimum delay can be used as an indicator of each link's current latency class (see Section 5.3.2). The propagation of a packet between a pair of geographic locations is limited by signal propagation time and the cumulative delay introduced by the intermediate network elements. Delays introduced by network elements are additive (the more elements the greater the delay) and a component that depends upon the current network's utilization. In contrast, the physical propagation delay is constant (for a given path) and depends upon the physical distance along the media between the source and destination multiplied by the propagation velocity in this medium. Certain changes in network routing result in an increase or decrease in the cumulative length of the physical path that packets have to travel, hence affecting the minimum achievable delay in the network.

Based upon the above, we argue that a change in the observed *minimum* delay represents a change in the path, while short term latency variance is due to competing traffic at the outgoing links. As a result, short term latency variance is *not* correlated with routing changes. In the next section we attempt to exploit knowledge of this minimum delay and introduce the FR algorithm.

5.4.3 FR Algorithm

In this section we present the FR algorithm which attempts to identify changes in latency classes by sampling the network delay at run-time. At the core of this approach we use the minimum observed latency (in a past window of samples).

Following our previous observations, we use a small subset of the latest values (i.e., front runners) in the reservoir to identify a change.

At a high level, the algorithm consists of three steps. First, using a sliding window we smooth the minimum observed latency over the latest set of samples giving us an estimate of the current minimum latency. Next, using a separate sliding window we compute the variance of the minimum latency over this window of samples. Based on our observations, for a single latency class, the minimum latency stays relatively stable. Finally, we monitor changes in the variance of the delay relative to the minimum latency and use this variance as an indicator of a change in a latency class.

Consider a one hour latency trace between Frankfurt and Tokyo (see Figure 5.7). This trace exhibits many routing changes (based on our traceroute measurements) and has frequent changes in latency classes. The bottom plot shows the variance in the minimum latency for each of the sliding windows. The vertical red lines indicate the moments when a change in latency class is identified by FR. In this example we used a sliding window of 10 samples to compute both the minimum latency and the variance in the minimum latency. This configuration was selected manually. In this example, FR was able to identify all changes in latency classes, with zero false positives. The average delay in identification was under 20 latency samples. This delay can be reduced by prior knowledge of all possible latency classes on a given network path. The proximity of an observed minimum latency to a known latency class can be used to reduce the number of samples required to identify a change.

Excessive congestion can affect the notion of the minimum latency (see Figure 5.6b) and prevent FR from clearly identifying the current latency level, which could lead to false positives of changes in latency levels. Highly congested intervals can hide a number of possible changes in latency classes and it is impossible to simply use latency measurements alone to identify these changes. Traceroute can be used to identify the current network path, hence consecutive mapping of this network path to a previously observed latency class can be used to assist FR. However, related work suggests that congestion is not the norm for the Internet [59], thus we expect that FR could be utilized most of the time.

By exploiting knowledge of latency classes obtained by performing traceroute (see Section 5.3.2) or inferred by using the FR algorithm on the latency stream, we can isolate latency due to congestion from latency changes due to change in network path. Using the residual latency signal gives better estimates of network variance and better indicates potential congestion.

The approach taken by the FR algorithm is very specific to the architecture of the modern WANs and cloud computing infrastructures. The main advantage of this approach is that it takes into account the underlying infrastructure and

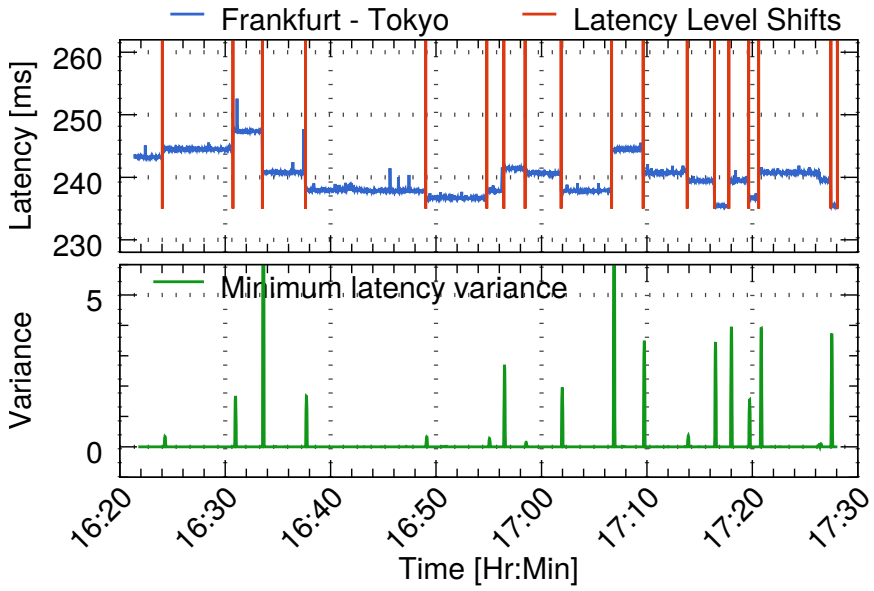


Figure 5.7: The top plot demonstrates RTT latency trace (blue line); vertical lines indicate latency level shifts as identified by FR. The bottom plot shows the variance in minimum latency computed over window of 10 samples FR can identify all routing changes on this trace.

its behavioral characteristics. Moreover, FR is more suited for tracking network conditions than generic stream sampling techniques, thus FR is a first step in tracking changes in network conditions while taking advantages of knowledge of the underlying infrastructure.

5.4.4 Estimating Congestion

In this section we show how the knowledge of latency classes can be used to understand the network's state. Figure 5.8 shows three sets of plots. The top plot **(A)** shows a one hour of RTT latency trace collected between Frankfurt and Singapore. Note, this sample contains a few distinct latency classes, which visually shifts the latency trace up or down along the Y axis. Conventional latency tracking methods, such as EWMA or reservoir's median are unable to clearly distinguish between changes in latency class (caused by routing changes) and latency variance (caused by competing traffic).

By exploiting knowledge of latency classes obtained by performing traceroute (see Section 5.3.2) or inferred by using the FR algorithm on latency stream, we can differentiate latency due to congestion from latency changes due to change in network path. Graph **(B)** in Figure 5.8 shows the residual latency after we subtracted the latency of each network path. This graph simply shows the variance

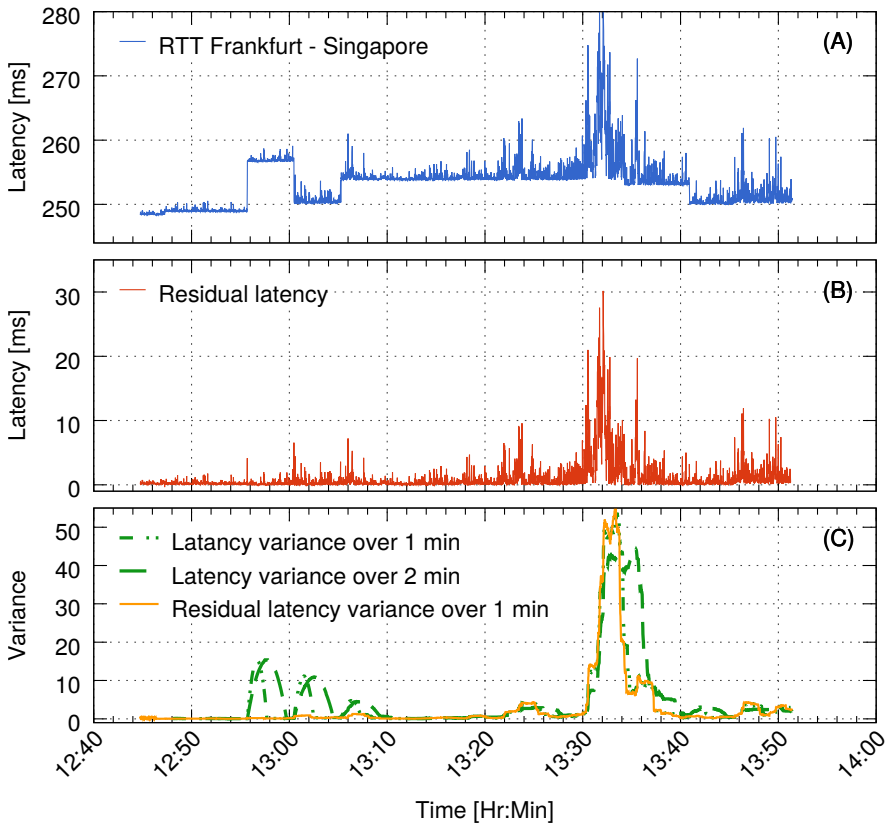


Figure 5.8: One hour RTT latency trace from Frankfurt to Singapore is shown in the top plot. The middle plot shows the residual latency after we subtracted the base latency of each network path. The bottom plot shows latency variance computed based on the samples from the top (green dashed lines) and middle (yellow line) data sets.

from the base latency of each network path.

By analyzing the residual latency we can clearly identify intervals of congestion, while ignoring changes in latency due to routing changes. At 13:00 in Figure 5.8 **(C)**, we can see spikes in variance, computed over the initial latency sample. Note, these elevations are longer when the latency variance is computed over the longer window of samples (2 minutes). These elevations in variance are false positive indications of potential congestion, as we can see that there was no corresponding residual latency. Note, that the variance graph computed over the residual latency (yellow graph) is not affected during this same time. Between 13:30 and 13:40 both graphs demonstrate equal variance as expected. Therefore, we conclude that using the residual latency signal gives better estimates of network variance and better identifies potential congestion.

5.5 Evaluation

In this section, we evaluate EdgeVar against alternative latency measurement and network path characterization techniques using controlled experiments and trace replay. In order to perform this evaluation we obtained real world latency samples from our measurements probes across 10 geographically distributed datacenters of Amazon EC2 (as previously described in Section 5.3). We measured RTT delay among all pairs of datacenters, using application-level UDP pings. Measurements were performed twice per second and all of the VMs' clocks have been synchronized using each datacenter's local NTP server.

First, by using trace replay we feed latency samples into a subset of selected algorithms in order to evaluate FR's ability to track changes in latency classes. Next, we evaluate EdgeVar's ability to choose preferable network paths and compared EdgeVar to conventional techniques, such as EWMA and EDR.

5.5.1 Front Runner Evaluation

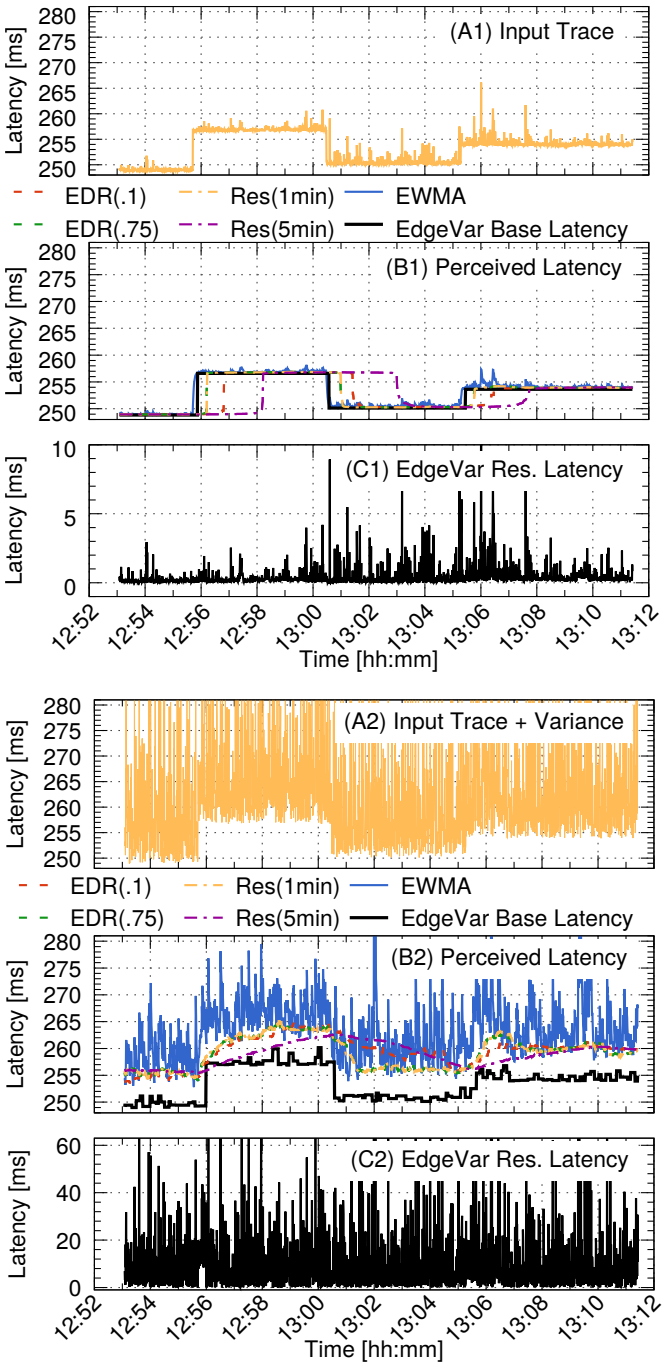
We begin by evaluating FR's ability to detect changes in latency levels and track two components of network delay: the base latency and residual latency. We compare our implementation with conventional latency estimation techniques that rely on EWMA, the median of a reservoir and Exponentially Decaying Reservoir (EDR)[‖] (currently used in Cassandra's dynamic replica selection algorithm). In this evaluation we used EWMA with the coefficient $alpha = 0.2$, the median value of static reservoirs containing 1 and 5 minutes of the most recent samples (labeled Res(1min) and Res(5min) in the figures), and EDR containing 1 minute of samples with adaptation coefficients of 0.1 and 0.75 (a larger coefficient indicates a faster adaptation rate). FR was configured as described in Section 5.4.3 in all of the following experiments.

Using the controlled trace replay environment, we performed a sensitivity analysis. We start by selecting a low variance latency trace that contains routing changes and use it to evaluate selected techniques. Then we repeatedly increase the latency variance in the input trace[**] and repeat our evaluation to better understand the effect of latency variance on the precision of FR and alternative techniques.

Low Variance Latency Trace The top plot (A1) in Figure 5.9a shows the low latency variance trace that we used as the initial input. Distinct shifts in latency levels are visible at 12:56, 13:01, and 13:05. We fed this trace into each of the latency estimation techniques. The middle plot (B1) contains the estimated value

[‖]We modified the Scales library [60], a python based implementation of the EDR [46] to support discrete time intervals.

[**]We adjust latency samples in the original trace by sampling from Pareto distribution with different coefficients.



(a) EdgeVar's comparison with alternative latency estimation techniques.

(b) Latency variance has been artificially increased by sampling from a Pareto distribution.

Figure 5.9: Two sets of plots demonstrate the ability of FR to track changes in latency levels under different network conditions. The top plots (A1 and A2) show the input latency traces used in each evaluation. The middle plots (B1 and B2) demonstrate network latencies as perceived by each estimation technique. The base latency level identified by FR is shown with a black line. The bottom plots (C1 and C2) show the residual latency extracted by FR.

of latency as perceived by each conventional latency estimation technique in this evaluation. In contrast, FR produces two outputs by decoupling the latency input signal into base latency (shown as a black line in B1) and residual latency (shown as a black line in C1). By looking closely at (B1) at 12:56 we can count the number of samples that it took for each technique to switch to a new latency level (from 250 ms to 257 ms). This evaluation is shown in Figure 5.10.

Each system showed a delay in identifying the shift. EWMA demonstrates the fastest adaptation rate by requiring only 8 samples; this result is explained by EWMA's exponential coefficient of 0.2 in this experiment. Both reservoir-based techniques took half of the reservoir's number of samples to shift the median latency to the new class (see Section 5.4.1). The high adaptation rate of EDR (with a coefficient of 0.75) prevented it from discarding any of the new samples, making it perform similarly to the reservoir-based techniques. EdgeVar performed well by requiring only 20 samples to realize that there was a change in latency class. This is 3 to 15 times faster than the reservoir-based techniques that have been used for comparison in this evaluation. Note, that this interval could be further reduced by exploiting knowledge of previously seen latency classes or by performing proactive traceroutes when FR suspects there may be a change in latency level.

High Variance Latency Trace Next, we used a Pareto distribution with $shape = 4$ and $mode = 30$ to produce a high variance trace. This trace is shown in Figure 5.9b (A2). At 12:56, in subplot (B2) it took 35 samples for FR to identify the change in latency class. In contrast, no alternative techniques considered in this evaluation were able to achieve this rapid detection of a change in latency class.

The high variance significantly reduced the precision of latency estimation of the reservoir based systems. These techniques can only maintain a general trend with a variable delay based on their reservoir size. However, EWMA showed the worst performance by overreacting to every single change in latency. The presence of alternative network paths during such an interval would result in excessive path flapping. In the next section, we demonstrate such a case using a different latency trace.

This evaluation shows that EdgeVar was able to operate successfully in a highly congested environment with no change to its initial configuration (i.e., parameter settings). Moreover, it maintains rapid adaptability during low congestion and gives a clear view of the network's state during the highly congested intervals.

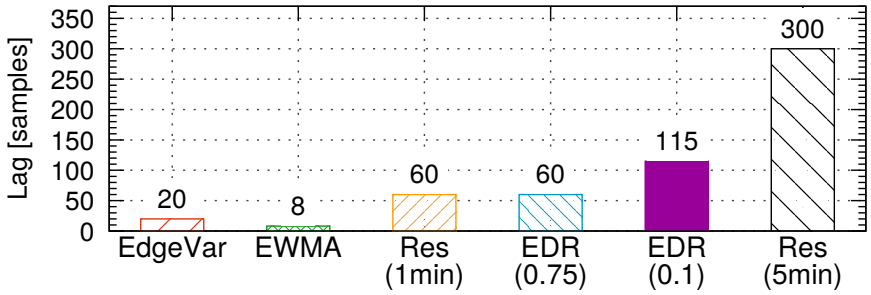


Figure 5.10: Number of samples that were required for each latency estimation technique to realize a change in latency level. Based on the low variance trace shown in Figure 5.9a.

5.5.2 Evaluation Trace Replay

In this section, we quantify the efficiency of EdgeVar by using the motivational trace presented in Section 5.1 in Figure 5.1b. Using two alternative network paths shown in the trace and the same set of latency tracking techniques, we compared EdgeVar's ability to select preferable network paths. In contrast to the previous section, we configured each algorithm to re-evaluate its network path selection every 10 seconds. This interval was chosen to prevent excessive network path flapping, which is undesirable in a real world system.

EdgeVar applied FR to each network path of the trace to extract base latency levels and residual variance (not shown). Unlike single-valued latency estimation techniques, EdgeVar utilizes both the base and residual latencies to better understand the network's state. Using these characteristics, EdgeVar computes and assigns a score to each network path and uses it to select a preferable path. The network path with the lowest score is chosen when a decision is made. The current implementation uses the following formula: $score = B + \alpha * V(w)$, where B is base latency, V is the variance of the residual latency computed over a window of w samples, and α is a variance sensitivity coefficient[††]. The coefficient α reflects the sensitivity of an application to latency variance. A higher value of α will "shorten" the width of a CDF of the request completion time, i.e., bringing the front and a tail closer together, leading to more stable performance (if there are alternative network paths to choose from).

By increasing α, an application will sacrifice the median of request completion time to reduce the tail of the distribution. In our evaluation we found $\alpha = 4$ and $w = 5\ minutes$ to perform reasonably well over different latency traces. These values have been selected manually (we leave auto-tuning for future work).

[††]Initially, we had an additional component of the median of the residual latency; however, we found it to be redundant in the computation of the perceived latency. The additive component of the median was replaced with an adjustment of the α coefficient.

Moreover, the final configuration decision has to take into consideration the particulars of a specific system and its desired QoS.

In Figure 5.11, EdgeVar is evaluated based on the trace replay shown in Figure 5.1b by demonstrating the CDF of the latency when using each of the evaluated network path selection techniques. We use minimum obtainable latency, i.e., the minimum of two network paths at each sample point (shown in the thick blue line) as a baseline comparison for other techniques. This figure indicates that EdgeVar closely follows the minimum tail of the achievable delay, while alternative techniques lag behind. The reason why EdgeVar follows the tail so well is that in the period from 13:00 to 16:00 (see Figure 5.1b) EdgeVar was able to distinguish the presence of a high latency variance for network path P1 and thus it persistently preferred the alternative network path P2. It is interesting that P1's latency level is lower than P2's, but P1's high variance makes P2 a better choice. All other techniques lacked this information, hence they were constantly changing paths and as a result missed using the better network path. The numbers of times each method changed its network path preferences are shown in Figure 5.12. While EWMA demonstrates the shortest latency tail among common techniques, it makes 4 times more changes among network paths than reservoirs and EDR and over 40 times more than EdgeVar. In contrast, by choosing the network path with the lower variance for the prolonged period, EdgeVar sacrificed 5 ms in the median latency compared to other techniques.

Increasing the frequency of network path selections is not a viable solution in practice. Today, it is common for applications to execute 1000s of requests per second (as opposite to the evaluation above that made the equivalent of 2 requests per second), as a result utilizing a congested link is bound to produce an even longer tail in practice. We are inclined to believe that performing latency estimation across multiple network paths at such a high rate and re-executing higher level logic (such as replica selection) is unacceptably wasteful and impractical. In this section, we demonstrated that by exploiting implicit information about the network's state, EdgeVar was able to achieve a shorter latency tail at the cost of increased median latency, while avoiding frequent network path flapping.

5.6 Limitations

This section summarizes the list of EdgeVar's current limitations. First, in order to function effectively, EdgeVar needs to build up a database of network paths and match them to unique latency classes. While the total number of network paths between any two geo-distributed datacenters is finite, the process of building a

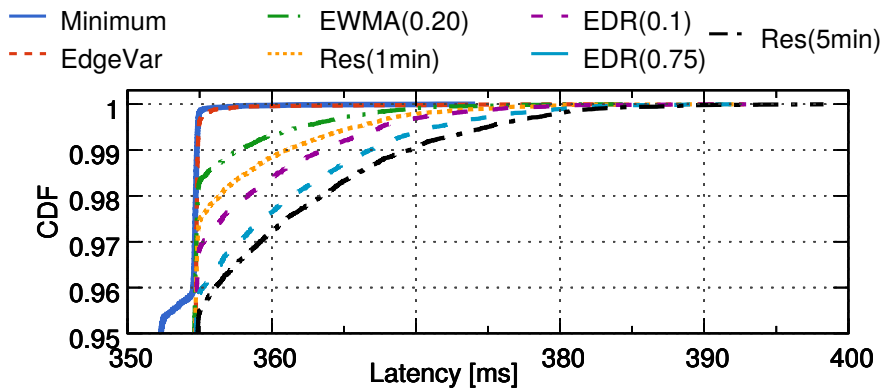


Figure 5.11: CDFs of latency obtained using each evaluated network path selection technique. EdgeVar (red line) closely follows the tail of the minimum achievable delay (blue line) (last 4%), while alternative techniques are lagging behind.

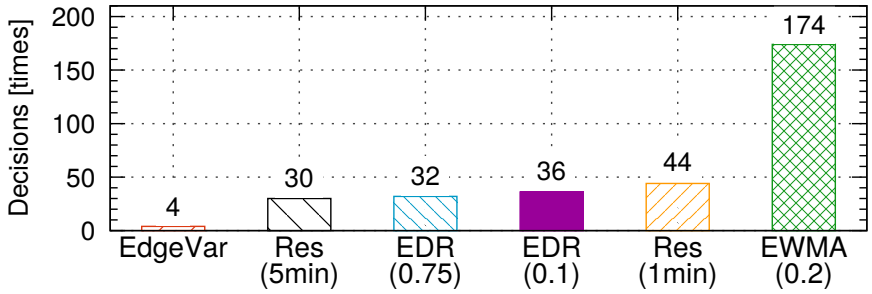


Figure 5.12: The number of times each network path selection technique changed its path preference during the trace. EdgeVar makes between 6 to 40 times fewer changes between network paths while demonstrating a shorter tail. Min. possible latency (not shown here) was achieved using 1890 dynamic changes between network paths.

comprehensive view of network paths might take some time[‡‡], depending on the frequency of routing changes in the network where EdgeVar is deployed.

Moreover, even after a long-term deployment of EdgeVar, it is possible to encounter a network path that was not previously seen. In that case, the current implementation of EdgeVar will not be able to match such network path to an existing latency class (even if such latency class has been previously observed). Thus, EdgeVar will rely on FR to detect a potential change in the latency class.

Next, similarly to the `traceroute`, EdgeVar relies on TTL notifications to be sent back by network devices (routers, switches, etc.). However, a cloud provider

[‡‡]Based on our observations in EC2, the most popular network paths were found within a few days after deployment of EdgeVar.

might choose to configure its network devices not to send TTL notifications. As a result, this will prevent EdgeVar from building a database of network paths and matching a current path with a latency classes. In that case, EdgeVar will have to rely completely on FR to identify changes in latency classes, which will affect the effectiveness of EdgeVar.

Finally, EdgeVar requires a set of measurement probes to be sent out periodically. While being a very small overhead, propagating probes along WAN still consumes network resources and marginally increases the cost of running a service.

5.7 Summary

In this work we proposed EdgeVar, a network measurement and classification tool, that takes into consideration the underlying network's infrastructure. We proposed a practical approach to decompose noisy latency samples into two components: base latency and residual latency variance. Exploiting this information enabled us to separate the effects of changes in network paths (presumably due to changes in routes) from congestion. We show how this information can be used to prefer network paths that lead to shorter tail latency at the cost of small increase in median latency.

Chapter 6

Reducing Long Tail Latency using Tectonic

In Chapter 5 we demonstrated the techniques for decoupling routing and congestion delays. The work presented in this chapter, is the logical continuation, that looks at the delays associated with packet's propagation time through the Linux networking stack and requests' service time.

In this work, we design, develop, and evaluate Tectonic, a load and latency monitoring system for geo-distributed services. We combine network path classification and traceroute techniques developed in EdgeVar with application packet timestamping in user and kernel spaces. This allows Tectonic to trisect request completion time into three components: base network latency, residual latency variance, and service time on the remote host.

We performed detailed study of packet propagation times in the Linux kernel on bare metal deployments and in virtualized EC2 instances. Using this data we established the relationship between the load inflicted on a server and the request propagation and service times. Then, by analyzing intervals between arrivals of consecutive packets we inferred the scheduling policies utilized by EC2 hypervisors.

This chapter represents an ongoing work and outlines our progress to date towards the goals of this thesis.

6.1 Introduction

When performing replica or server selection, distributed systems generally rely either on network latency alone, or on the request completion time which incorporates both network latency and service time. However, delay contributions from network latency and service time exhibit different properties (i.e., delay distributions) and change independently.

In Chapter 5 we introduced the concept of latency classes and demonstrated that in the absence of congestion, WAN latency is stable and depends on the routing. A typical geo-distributed service alone is unlikely to saturate bandwidth capacity of an inter-datacenter cross continental WAN link and therefore, is unlikely to cause network congestion. Thus, we can assume that network latency class is independent from the load (e.g., the number of clients' requests) of a single service. Moreover, latency class can be measured using techniques introduced in EdgeVar, and be provided explicitly to the service.

In contrast, a service time of a request is greatly depends on the load on the machine which serves the request. Naturally, two delay distributions associated with network latency and request service time exhibit different shapes, moreover, over time these distributions change independently of each other. Therefore, simply taking the median of the total request completion time (i.e., network latency + service time) does not accurately represents the delays encountered by the requests. In this work we argue that these two components of a request completion time (network latency and request service time) should be treated separately.

When communicating across WAN, network latency can have a dominant effect on the response completion time which in turn can lead to replica choices based on geographic proximity or the quality of the network path *without* consideration of service times at the different alternatives. If done naively, this can result in the majority of replicas choosing a single system. For example, in the case of Cassandra based service deployed across all geographic locations of Amazon's EC2 the distribution of requests directed towards each datacenter is *not* uniform. Figure 6.1 demonstrates that when a Cassandra based global service operates at the consistency level 5 (i.e., each Cassandra server attempts to select an additional Top-4 replicas to fulfill its consistency requirements) all of the servers will choose California as one of the replicas leading to this server being overloaded, hence *increasing* request completion time and variance.

The aim of this work is to provide a clear perception of individual latency components to allow applications to express their latency preferences and make conscious decisions, in a way that was previously impossible. For example, in a geo-distributed application such as Cassandra a service provider can now say: "I want to reduce my service tail latency and I am willing to sacrifice up to 20 ms in my median requests' completion time". In contrast, the lack of knowledge of these components can lead to suboptimal performance, increased tail request completion times, and oscillating choice of replicas.

The rest of this chapter is structured as follows. In Section 6.2 we describe the architecture and the use case of Tectonic. Next, in Section 6.3.1 we evaluate the overhead of using Tectonic based on our local deployment. In Section 6.3.2

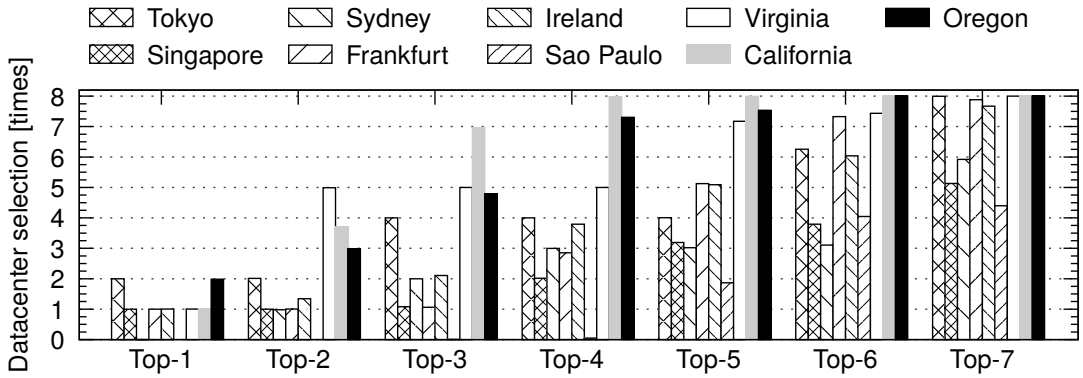


Figure 6.1: Replica choices based on network latency alone (one day trace replay). The X axis indicates the number of additional geo-distributed replicas that each server had to choose (i.e., the consistency level). The Y axis indicate the number of forwarded request handled by each server. For example, for a "Top-4" all servers choose the datacenter in California while no-one choose the datacenter in São Paulo.

we repeat our set of measurements on the EC2 and demonstrate correlation between propagation delays and the load on the server. In Section 6.4 we look at propagation delays associated with the hardware virtualization present in clouds and demonstrate the technique of inferring CPU scheduling policy utilized by XEN hypervisor. Next, in Section 6.5 we demonstrate how these scheduling policies can be inferred by analyzing packets' timestamps provided by Tectonic. We conclude this chapter in Section 6.6.

6.2 Tectonic

In the following sections we describe Tectonic and its implementation. In Section 6.2.1 we outline Tectonic's architecture and its scope of use. Next, in Section 6.2.2 we describe application message timestamping using Tectonic's user and kernel level modules and then demonstrate an example of using Tectonic to timestamp Cassandra's requests and responses. Finally, in Section 6.2.3 we describe Tectonic's tracerouting and network path classification.

6.2.1 Architecture

In this section, we describe Tectonic's architecture. Tectonic is implemented as a kernel space process that runs in the operating system (OS) where an application is deployed. A single instance of Tectonic can potentially serve more than one application running on OS. Tectonic performs network measurements,

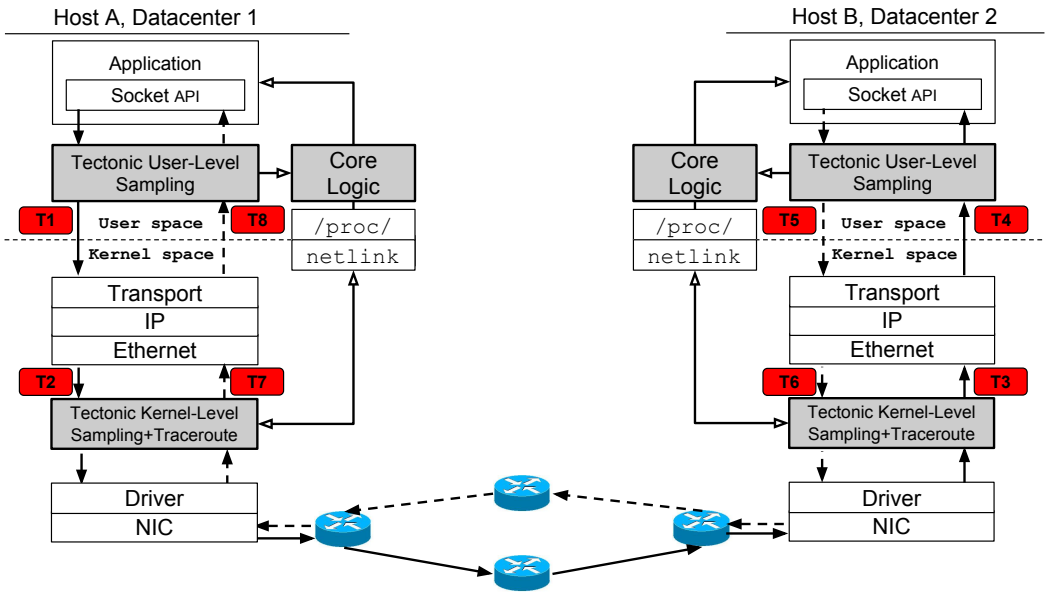


Figure 6.2: Tectonic's networking architecture. Filled solid (dashed) line arrows indicate propagation of requests (responses) sent by the application running on the host A (B) towards the application running on the host B (A). Hollow solid line arrows indicate Tectonic's internal communication. Red square boxes indicate locations where Tectonic performs requests (responses) timestamping. Details for virtualized deployments are omitted.

application request timestamping, exchanges information with other Tectonic instances (running on other hosts), and shares digested network and timestamping information with applications via the `/proc/` filesystem. Figure 6.2 outlines Tectonic's integration into the OS where a distributed application is deployed. Tectonic's elements are highlighted in grey filled bold bordered boxes and are comprised of 3 main elements: (1) a user level socket API library, (2) a kernel level sampling and traceroute modules, and (3) Tectonic's core logic.

First, Tectonic performs user space and kernel space timestamping of the targeted application's messages. Each instance of Tectonic timestamps incoming and outgoing requests and responses exchanged between distributed instances of the application. The timestamping is performed in user space at the socket API level and in the kernel space just before packets are sent to the NIC. The timestamp checkpoints are shown in Figure 6.2 as red boxes labeled from t_1 through t_8.

The timestamp information is then exchanged among Tectonic's instances allowing matching of sent and received requests and responses. Therefore, the propagation time of the application requests is recorded at each processing stage.

This gives a complete view of where requests (and corresponding responses) spent their time and is used by Tectonic to decouple network latency from the service time latency (both at the kernel and application's levels). Recognizing messages on a TCP stream requires use of application domain knowledge (i.e., logical message structure and communication ports), but does not require any modification of application code.

Second, using the kernel level networking layer, Tectonic performs traceroutes along the path of the application communication. Using the 5 tuples of the application flow (source and destination IP address, source and destination ports, and transport protocol) the system crafts network packets (probes) that match the 5 tuples of the application's traffic. Imitating the application layer traffic is important to ensure that network probes follow the same network path as the application's traffic [55]. We utilize unused header fields (such as the Urgent Pointer) and specific sequence numbers to distinguish Tectonic's traceroute packets from the application's traffic.

The **Third** component is the Tectonic's core logic that performs network path classification and maintains a mapping of the existing network paths to distinct latency classes. This component is implemented as a user space process. Using *netlink* sockets Tectonic's kernel modulus share the available tracerouting information (sent probes, received ICMP Time Exceeded packets and returned packets that reached the remote Tectonic's module) with the user space network path classification component. Using this information it builds the database of network paths on each link, performs network path classification, and then utilizes this information to decompose network latency into the base latency associated with the network path taken by the application's packets and the residual latency variance due to competing network traffic. Utilizing Tectonic's `proc` filesystem it exposes the run-time information about the base network latency and residual latency variance on each link.

The **fourth** module (not shown in Figure 6.2) is application specific code that takes advantage of the network measurements and triggers application decisions. This module substitutes application's network measurement functionality by reading the relative data directly from the `proc` filesystem. This is the only module that requires application source code modification. In the case of Cassandra, this module has been implemented as a modification to the dynamic endpoint snitch in Cassandra's replica selection logic.

6.2.2 Application Message Timestamping

Tectonic's message tracking functionality performs timestamping of an application's requests and responses in both user and kernel spaces. This timestamping covers the entire lifespan of a request, i.e., starting when the request is sent and ending

when the corresponding reply is received. Timestamps from remote hosts are exchanged asynchronously from application traffic and are used to compute the complete propagation history of each sampled packet. Tectonic performs selective message sampling to minimize its negative impact on performance.

6.2.2.1 User Space Sampling

User space sampling is implemented via Tectonic's Linux socket API wrapper library. Tectonic uses dynamic linking to selectively replace the normal Linux socket library with a custom wrapper library. Using the `LD_PRELOAD` environment variable at the application's boot time, the new library is preloaded before all other libraries (see Appendix A.2). Therefore, when the target application performs a socket API call, it calls Tectonic's specific code. This allows for application agnostic traffic monitoring as there is no need to modify the application's source code. The wrapper library performs the following operations: (1) it monitors the events of opening and closing network sockets by the application and maintains a mapping of IP addresses to integer socket file descriptors, thus differentiating among multiple senders and receivers; (2) it forwards all of the application's socket calls to the kernel, thus allowing the application to perform its natural network communication seamlessly; and (3) by intercepting the application's send and receive calls Tectonic timestamps requests and responses traveling downstream and upstream respectively. Using knowledge of the application's message structure we extract unique message identifiers, such as request and response IDs.

The extraction of message identifiers is a O(1) operation as we read the message ID as a set of bytes at a specific offset in the message buffer. This approach relies on the assumption that the start of the logical message is aligned with the send or receive buffer, which holds for the message-oriented protocols, such as UDP.

However, for byte stream oriented protocols (e.g., TCP) message boundaries are not preserved. Two cases are possible: (1) if the start of the logical message is aligned with the start of the receive (send) buffer we read the size of the logical message and note the size of the buffer. Thus, if the buffer does not yet contain the entire message we postpone timestamping until the application receives (sends) enough bytes to match the logical message size. (2) If the start of the logical message does not align with the buffer, this buffer is ignored.

To minimize the overhead of reading application specific message IDs, Tectonic does not perform deep packet inspection by searching through the byte stream to identify logical message boundaries. During our evaluation in Sections 6.3.1.2 and 6.3.2.2 we note two observations that suggest that it is sufficient to look only at the buffers aligned with the start of the application's logical messages. First, applications operate on the message level, thus when a `send` call is executed, the application sends the complete message to be send, thus the majority of the

downstream traffic will be buffer aligned. This occurs because applications (such as Cassandra) do an explicit PUSH to force the sending of a TCP segment containing a request or response, hence case (1) occurs. Second, during our measurements on EC2, we noticed only a small amount of TCP fragmentation both at the network and kernel levels. We observed that in a large number of cases, TCP segments correspond one-to-one to application messages. Since Tectonic uses sampling, even a small fraction of unfragmented packets is sufficient to build a global view of message propagation delays.

6.2.2.2 Kernel Level Timestamping

Kernel level timestamping is implemented as a kernel module. Using Netfilter's kernel hooks `NF_INET_PRE_ROUTING` and `NF_INET_POST_ROUTING`, the module is capable of intercepting incoming and outgoing packets close to the network driver level. Using application knowledge (specifically the transport protocol and destination port number*) the system's kernel module isolates application traffic by examining the `skbuff` data structure.

Next, we use a user space sampling to extract unique message IDs from the packet's payload. This kernel space sampling, allows us to timestamp the application's messages just before they are sent to the NIC or just after arrival. Comparing kernel and user level timestamps we can compute the time it takes for an application's messages to traverse the Linux networking stack and thus, we can isolate the request's service time.

6.2.2.3 Sharing Timestamps

The user and kernel level timestamps along with the associated message IDs, are asynchronously shared with remote instances of Tectonic. The relevant information is propagated out-of-band via a separate TCP connection maintained among pairs of Tectonic instances. Out-of-band communication was selected for the following two reasons: First, sending this traffic out-of-band is less likely to cause the application's data to be fragmented at the network level (as we have not changed the number of bytes passed by the application over the application's socket(s)). Second, encoding control messages in-band by appending the information to UDP packets or encoding information into a byte stream would require an additional copy operation at the sampling module before passing the native message to the application, which would lead to additional overhead.

Both the user level and the kernel level sampling modules introduce a small additional delay between 1% and 2% (15 μs) to the minimum message propagation

*It is sufficient to know only the destination port number, as the application uses a single port for communication with its peers.

time while having negligible effect on the median and tail percentiles. In Section 6.3.1.2 we evaluate the effect of packet sampling on the application's performance.

6.2.2.4 Timestamping Cassandra's messages

Cassandra nodes use TCP for inter-node communication. In this section, we begin by describing Cassandra's logical message structure. Then we demonstrate how this application specific knowledge is used by Tectonic to track Cassandra's requests and responses propagation times.

Cassandra utilizes a set of ports and protocols for various needs (JMX, Thrift client API, CQL transport, etc.). In this section, we concentrate on Cassandra's inter-node communication performed over TCP on port number 7000 (i.e., to the destination port number 7000). To identify a message we rely on 3 distinct message fields: Cassandra's magic number, message ID, and operation type. All 3 fields are 32-bit integer numbers located at the 1st, 4th, and 17th-byte offsets in the message[†].

First, Cassandra's magic number is a constant message boundary value equal to `0xCA552DFA` and is present in all messages. This value is used by Cassandra as a sanity check to validate senders. We utilize this number to identify when the start of the logical message is aligned with the start of the buffer read from the socket or TCP segment. The presence of this number at the expected offset indicates the beginning of a logical message.

Each instance of Cassandra uses unique message identification numbers to map its requests to responses. This number is an integer counter that starts from zero at the application's start time and is incremented for message sent. This counter is shared among data communication and control messages exchanged among Cassandra's nodes. When a replica generates a response it put the request's message ID into the response header. This allows matching requests of one replica with the corresponding responses received from another instance. The source and the destination IP addresses are used to differentiate responses from multiple replicas.

Finally, we identify the operation ID (in Cassandra also known as a *verb*). Read request and read response correspond to the values of 3 and 4 respectively. The complete list of operations can be found in Cassandra's source code in `MessagingService.java`.

Specifically for Cassandra, we noticed that the majority of message concatenation occurs at the application level in Cassandra's `BufferedDataOutputStreamPlus` data structure, which acts as a custom buffer in front of the socket. Note, in

[†]Based on Cassandra v3.7.

contrast to the send operations performed by Cassandra, application level reads attempt to retrieve a buffer of 4096 bytes per socket call. The message level parsing is implemented at the application level.

6.2.3 Traceroute and Network Path Classification

At the high-level, Tectonic's tracerouting functionality operates in three phases: network paths measurements, network paths classification, and run-time detection of changes in the current network path class.

During the **first** phase, Tectonic detects network paths between each pair of geo-distributed hosts and estimates their base latency (based on the minimum observed latency sample [61, 62]). For each pair of hosts, this is done by performing periodic synchronized traceroutes, such that both hosts initiate traceroutes at the same time[‡] in opposite directions (see the next Section 6.2.3.1).

Next, for each pair of hosts, multiple network paths are classified into path classes based on the equivalence of their base network latencies. Hence, a packet traveling along a path that belongs to a particular path class will incur the same (comparable)[§] base network latency as if the packet would travel along any other path in that class. Using this classification, for each pair of hosts, we identify unique IP addresses at the distinct hop positions from the source, along the path to the destination, that can be used to differentiate among observed path classes (see Section 6.2.3.2).

Finally, Tectonic performs active monitoring of such unique IP addresses in order to detect the moment when the current network path changes to a path that belongs to a different path class, thus resulting in a different base latency. When such a change is detected, the information about the new network path is made available for the application via the `proc` filesystem. This is done by performing partial traceroutes, i.e., traceroutes that contain only a subset of probes and designed to check the specific IP addresses at distinct distances from the source towards the destination (see Section 6.2.3.3).

6.2.3.1 Network Path Measurement Phase

In order to accurately measure the network latency between a pair of geo-distributed nodes, Tectonic utilizes the knowledge of network paths and their corresponding base latencies. A comprehensive view (a database) of the network paths among Tectonic's hosts is built over time by performing periodic synchronized traceroutes. The technique of correlating network paths with latency

[‡]Subject to NTP synchronization and Linux process scheduling.

[§]The similarity threshold is configured according to the application's sensitivity towards network latency.

is based on the EdgeVar's measurements previously described in Section 5.3 page 93.

Using the 5 tuples of the application flow, Tectonic crafts network packets to match those of the specific flows utilized by the applications. This is done to ensure that in the presence of per-flow or per-destination load balancing, Tectonic's traceroute probes will follow the same network paths as the application's traffic [55]. Hence, a routing change detected by Tectonic's tracerouting probes reflects a change experienced by the application's traffic.

First, we demonstrate how Tectonic performs synchronized tracerouting between a pair of hosts, then we describe how pairwise tracerouting is coordinated on the global scale of the geo-distributed system.

Per-flow tracerouting

Tectonic's tracerouting implemented as a kernel space timer system. This module is responsible for periodic generation and dispatching of traceroute probes. Traceroute functionality also relies on the Tectonic's sampling module (previously described in Section 6.2.2.2) to collect ICMP Time Exceeded and reply messages sent back by the network elements and remote instances of Tectonic respectively.

Figure 6.3 illustrates the timeline of a single round of tracerouting between hosts (A and B) in two distinct datacenters (1 and 2); arrows indicate the exchange of messages among Tectonic's modulus. The tracerouting is periodic and based on the timer with a preconfigured interval. Step #1 in Figure 6.3 (shown as a black hexagon) illustrates the beginning of the tracerouting. Using the signature of the application traffic Tectonic crafts a set of network packets by assembling the `sk_buff` data structures and dispatching them via `dev_queue_xmit` driver's queue towards the host B.

Unlike the classical traceroute that iteratively increments TTL values of its probes, Tectonic sends all probes in a single burst of packets[¶]. This reduces the completion time of a single round of the traceroute, allowing more frequent sampling and improves the precisions by lowering the chances of encountering routing changes in the interval between the first and the last traceroute probes. The maximum range of the TTL values for each path is determined by the maximum network distance previously observed on that path.

The ICMP Time Exceeded packets include IP header and 8 bytes of the original payload [64]. In the case of TCP, this includes the source and destination ports (2+2 bytes) and the TCP's sequence number (4 bytes). The TCP sequence number is not used in the flow identification, and thus it can be modified without affecting the routing of the packets [55, 65]. As it was previously demonstrated by Paris

[¶]A similar modification has been implemented in Scapy [63] a popular Python based networking library.

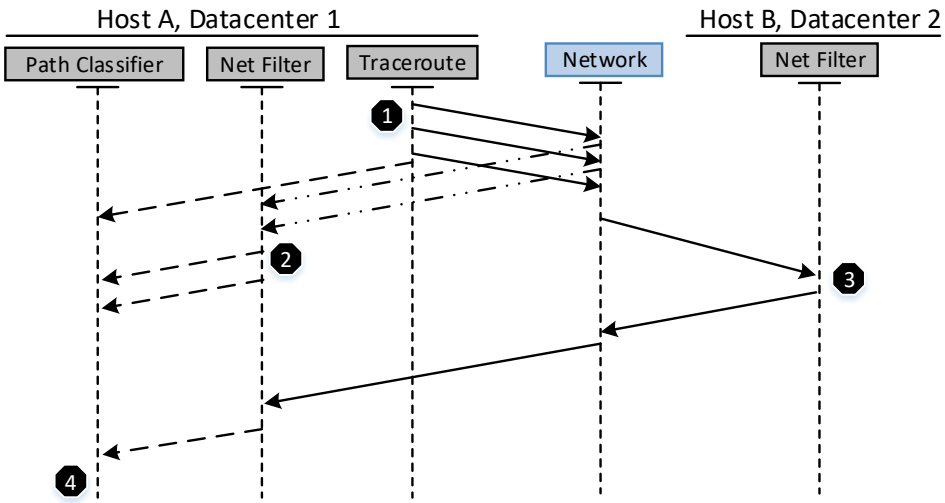


Figure 6.3: One round of Tectonic's tracerouting from the host *A* towards host *B*. Solid line arrows indicate traceroute probes with variable TTL value generated by Tectonic and the corresponding probe that reached the destination and returned back by the Tectonic's module running on the host *B*. Dotted lines represent ICMP Time Exceeded messages returned from the network. Dashed line arrows indicate kernel to user space communication via `netlink`.

Traceroute [55], by setting unique TCP sequence numbers we can match the sent probes with the returned ICMP Time Exceeded packets, and thus reconstruct the network path. Using `netlink` socket, the tracerouting module notifies Tectonic's network path classifier about the subset of probes that has been sent (indicated by a dashed line from Traceroute to Path Classifier module in Figure 6.3). This includes the destination IP, timestamp, and the mapping between TCP sequence numbers and TTL values of the dispatched probes.

In contrast to the traditional traceroutes, the process of dispatching probes is decoupled from the process of collecting ICMP replies and reconstructing the network path. Tectonic reuses the existing sampling module implemented as a Netfilter hook to collect ICMP Time Exceeded packets. Collected ICMP replies are forwarded to the Path Classifier module via `netlink` (shown in step #2).

The traceroute probes that reached the destination, are intercepted by the Tectonic's kernel module of the host *B*. Using lightweight zero-copy software classification we separate the application's traffic from Tectonic's probes that mimic that traffic. The application's traffic is forwarded unmodified to the Linux

networking stack. For the tracerouting probes, Tectonic rewrites the packet's header fields to match the application's traffic in the opposite direction, places a timestamp into the payload of the packet, and sends it back to the original sender (shown in step #3).

Similarly to the ICMP replies, the sampling module intercepts the reply packet and forwards it to the Path Classifier via `netlink` (shown in step #4). By combining the three sources of information (the sent probes, ICMP Time Exceeded replies, and the reply from the remote host), Path Classifier reconstructs network path in the forward direction (from the host A to host B). Moreover, using NTP traceroutes are synchronized among all pairs of hosts, thus the host B initiates traceroute towards A at approximately the same time as host A sends its probes towards host B. The reconstructed paths are then exchanged among two hosts (using out-of-band communication from the application's traffic) allowing each instance of Tectonic to obtain Network Path Pair with the exact RTT latency experienced by the packets (see Section 5.3.2).

Coordinated Tracerouting

Tectonic's pairwise tracerouting is also synchronized at the global scale among all hosts in the cluster. During the coordinated tracerouting, all hosts perform synchronized tracerouting towards a single target node (from here simply target). This produces a uniform view of network paths from the target node towards all other hosts.

Coordinated tracerouting is arranged in *epochs*. Each epoch incorporates tracerouting rounds (simply rounds from here on); one round per Tectonic's host. All rounds uniformly distributed along the length of the epoch; all epochs are of the same pre-configured length. Each round contains a set of per-flow traceroutes associated with a particular node in the system. During such operation, all hosts synchronously run the per-flow tracerouting towards a single target (see Figure 6.4), while the target host simultaneously performs traceroutes in the opposite direction towards all other hosts.

The coordination among Tectonic's nodes is based on the wall clock synchronization among the machines and maintained by NTP. Each node independently computes the start time of each epoch (based on its local wall clock time). The start time of the first epoch T_{zero} is computed as $(t_{now} - (t_{now} \mod L)) + L)$ where t_{now} is the local wall clock time and L is the length of the epoch. The start times of the consecutive epochs are computed by adding the multiples of L to T_{zero}. For a pair of hosts (A and B), Tectonic uses packets' timestamps to match traceroute probes sent from host A towards host B and vice versa. Thus, epoch numbers are not synchronized among the hosts; it is

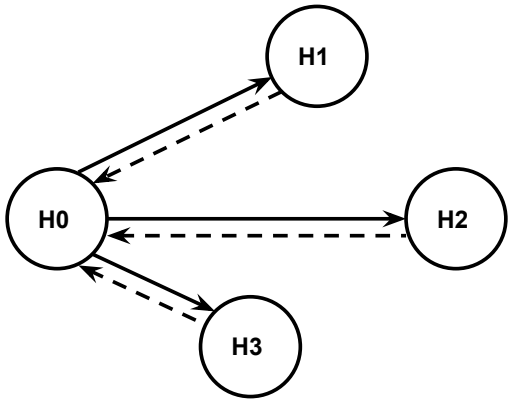


Figure 6.4: One round of the coordinated tracerouting, all hosts perform per-flow tracerouting towards the single destination (Host 0). Solid and dashed line arrows indicate traceroute in the forward and return directions.

sufficient that epochs on all hosts start at the approximately the same time[‖].

Next, the time of each traceroute round computed as an offset from the start time of the epoch and equal to $T_{zero} + L \times i + j\frac{L}{N}$, where N is the total number of Tectonic's hosts, i is the index of the epoch, and j is the index of the round. The target destination and the order in which hosts dispatch traceroutes is deterministically derived from the numerical order of the IP addresses in the cluster (all nodes of the Tectonic's cluster are known in advance). Thus, each host knows when it has to perform a traceroute towards a common target and when it should dispatch traceroutes towards all other hosts (i.e., the host is the target host in this round).

While the number of traceroute probes sent towards the destination is proportional to the number of hops (on average 16 hops between any pair of datacenters in Amazon EC2), the number of probes that will reach the destination and thus will require the host to process them, is small. Moreover, the difference in the WAN latencies among geo-distributed datacenters ensures that the traceroute probes dispatched from multiple hosts towards a single target will not arrive simultaneously. Hence, coordinated tracerouting will not produce a significant burst of incoming packets on a per host basis, thus the overhead of performing coordinated tracerouting is small and does not put an extensive load on the target node.

Coordinated tracerouting, allows obtaining a single snapshot of all routes towards a single destination. Sending probes simultaneously towards a single target

[‖]Based on the NTP synchronization.

increases the consistency of the obtained network paths and provides insights into the network path coalescence.

6.2.3.2 Network Path Classification Phase

Tectonic's Path Classifier performs grouping of network paths based on the minimum latency observed on each path. For each pair of hosts, this grouping produces a small number of network path classes where each path in its class demonstrates the same (comparable) base network latency. All unique network paths have distinct IP addresses at the specific hops by definition (see Section 5.3). Moreover, when network paths grouped into classes, the resulting path classes also have distinct IP addresses. This makes it possible to distinguish among different path classes at the run-time by monitoring the presence of a particular IP addresses at a specific hop position.

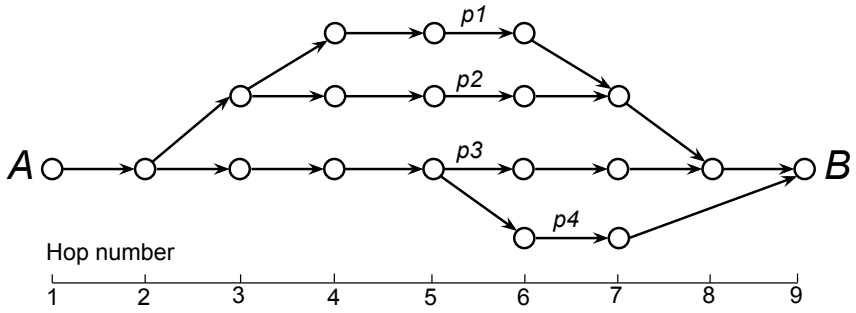


Figure 6.5: A set of four measured paths between hosts A and B. Horizontal ruler indicates the hop distance from A along the network paths towards B.

For example, Figure 6.5 demonstrates a set of four network paths between hosts *A* and *B*. Consider the scenario when these paths form two path classes (*p1*, *p2*) and (*p3*, *p4*). The IP address at the third hop position is the differentiating IP address for these path classes. Thus, by sending a single traceroute probe with the TTL value of 3 we can obtain the IP address of the network device located at the 3rd hop position from the host *A* and therefore identify current network path class between hosts *A* and *B*. Alternatively, if the two path classes will be formed as (*p1*, *p4*) and (*p2*, *p3*), we can differentiate among these classes by monitoring the IP address at the 6th hop position from the host *A*.

6.2.3.3 Network Path Class Change Detection Phase

For each network path class, Path Classifier computes the smallest subset of distinct IP addresses and their corresponding distances along the network path,

that differentiate the path class from other previously observed path classes. This set of IP addresses is used by the tracerouting module to perform *partial traceroutes*, i.e., for each IP address in the set, Tectonic crafts a packet that matches the application's traffic and sets the TTL value to match the distance along the path to the specific hop where this IP is expected to be.

Partial traceroutes have lower overhead than the full traceroutes from the path measurement phase and can be performed at a higher frequency to detect changes network path classes. Moreover, partial traceroutes do not require global or pairwise synchronization and are performed independently by each host, thus each host monitors changes in network path classes in the forward direction from itself to other hosts in the system. When a host detects a change in the network path class on a particular link, it informs the remote host; the remote host performs unscheduled traceroute along the return path to identify if the return path has been affected by the change.

6.3 Evaluation

In Section 6.3.1 we begin Tectonic's evaluation by measuring the overhead of packet sampling using physical non-virtualized machines. Next, in Section 6.3.2, we describe the evaluation performed on Amazon EC2 in which we measured the amount of time Cassandra's request spends in the Linux networking stack and user space.

6.3.1 Local Evaluation

In this section we evaluate the overhead of using Tectonic to sample and timestamp packets, at both the application and kernel levels. We begin by describing the setup in Section 6.3.1.1. Then, in Section 6.3.1.2, we evaluate the overhead of Tectonic's packet classification and timestamping by examining the case of Cassandra.

6.3.1.1 Local Setup

We conducted our sensitivity evaluation using three physical machines at our local premises. Each machine was equipped with a dual socket 8-core Intel Xeon CPU E5-2667 v3 operating at 3.20 GHz and 64 GB of RAM. Hyper-threading was disabled and the OS is Ubuntu 14.04 with Linux kernel v.3.14. All three machines were located in a single rack and directly connected via 1 Gbit physical interfaces. The RTT latency among all machines is roughly 0.15 ms.

The two servers were used to setup Cassandra's cluster. The remaining machine generated workloads using the YCSB benchmark (see Figure 6.6). We

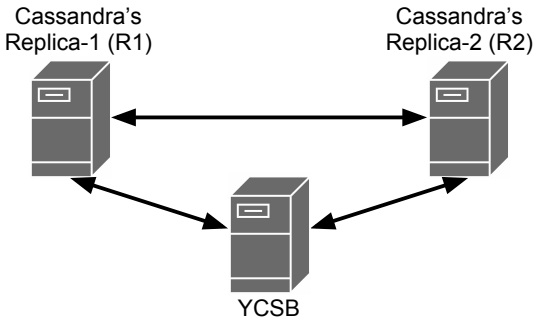
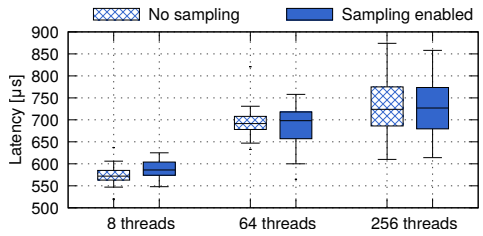


Figure 6.6: Non-virtualized testbed setup. Two physical machines (on the top) comprise the Cassandra cluster. The third machine generates a workload using the YCSB benchmark. The workload is equally spread over the two machines in the cluster. Tectonic was used to timestamp application level traffic between replicas 1 and 2.

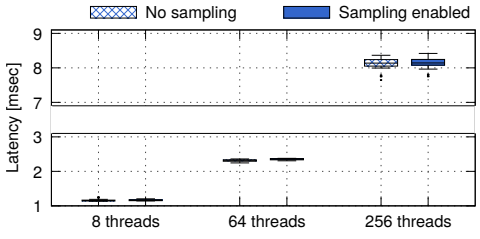
used YCSB [66] (v0.10.0), a popular storage systems benchmark, to generate a sample workload for our measures of the propagation time of requests and corresponding responses. YCSB is a closed-loop benchmark where each thread waits for its request to be completed before it sends additional requests. The Cassandra cluster was configured with the full data replication over the two machines (`replication_factor` of 2). Tectonic was used to monitor intra-cluster communication messages; YCSB messages were not timestamped. By default, YCSB employs round-robin load balancing by dividing its requests among all replicas of a Cassandra cluster.

6.3.1.2 Sensitivity Evaluation

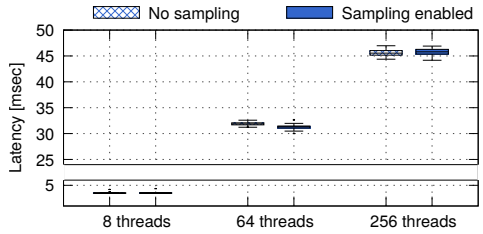
Using the setup described above, we measured request completion time while timestamping packets using three scenarios with different loads generated by YCSB benchmark. In particular, we used 8, 64, and 256 client threads to evaluate the effects of timestamping under low, medium, and high system loads respectively. In each load level, YCSB was configured to perform 1M, 8M, and 16M key reads respectively. This was done to equalize durations of workloads across different load levels. The database contained 10K unique keys, the relatively small number of keys guaranteed that all the data has been cached on both replicas. The read consistency was set to `TWO`, thus each replica upon receiving a request had to validate the data with the second replica *before* sending a reply. For each load configuration, we compared the request completion times reported by YCSB while running the cluster with and without Tectonic's sampling. Tectonic was configured to timestamp all application level messages and all incoming TCP segments (that



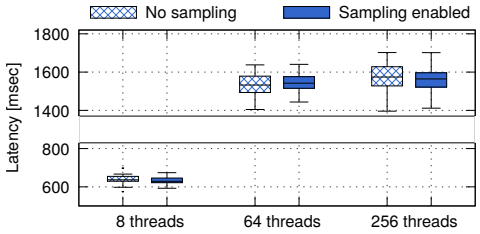
(a) Minimum request completion time



(b) 50th percentile of request completion time



(c) 99th percentile of request completion time



(d) Maximum request completion time

Figure 6.7: Tectonic's sampling overhead, measured under 3 different workloads. In all three cases the additional delay introduced by adding the instrumentation code is small. The bodies of the boxplots indicate the median, first, and third quantiles, whiskers extended for 1.5 interquartile range (IQR) in each direction, the points lying beyond whiskers indicated as dots.

align with the start of the TCP payload) on both machines. Overall, each application packet was timestamped at 8 locations (see Figure 6.2). We repeated each experiment 30 times.

Figure 6.7 shows four sets of box plots demonstrating the minimum, 50th, 99th, and the maximum request completion time reported by YCSB during the set of measurements. Each subfigure contains 3 pairs of box plots corresponding to 8, 64, and 256 client threads. The dashed box plots correspond to the baseline tests were two replicas were running without any of Tectonic's instrumentation, while the solid colored box plots correspond to the cases with the added instrumentation.

Figure 6.7a demonstrates the minimum request completion time under the three workload scenarios. Tectonic's sampling shows a small overhead. Over the 30 repetitions, the difference in the median of the minimum requests completion time among the three workloads is between 1% and 3%. This is equivalent to an average difference of 15 microseconds.

Looking at additional delay incurred at higher loads, the overhead of

performing timestamping is less than 1% of the baseline performance. The timestamping overhead is related to the amount of packet concatenation (i.e., coalescing) occurring in the TCP flows between the Cassandra replicas. Under a light load, the one-to-one mapping between TCP segments and logical messages is preserved for the majority of packets. At this workload, Tectonic is capable of timestamping almost 100% of all messages. However, at a higher workload, application messages become fragmented over TCP segments. Therefore, we observed a decline in the fraction of packets being timestamped, thus the per packet overhead of performing timestamping at the higher loads (64 and 256 client threads) is smaller than for the low load (8 threads).

In Figure 6.7c with 64 threads and Figure 6.7d with 8 threads and 256 threads test cases demonstrate a small improvement in the median completion time while using Tectonic's timestamping. This indicates that the overhead of timestamping is comparable to the precision of the request completion time measurement available using the YCSB benchmark.

Interestingly, the baseline and the Tectonic's instrumentation demonstrate nearly identical tails in their request completion distributions time. Therefore, we conclude that Tectonic's timestamping does not affect the shape of the distribution and gives a sufficiently precise view of the request completion times.

6.3.2 Cloud Evaluation Setup

In this section we move our experiments onto the Amazon EC2 cloud. Using Tectonic's timestamping techniques, we measure and evaluate the propagation time of Cassandra's requests and responses through kernel and user spaces. Then we compare our measurements with results obtained in our local premises on the bare metal installation.

6.3.2.1 Cloud Setup

We conducted our experiments using 3 identical `c3.large` EC2 instances, each with 2 virtual CPUs, 3.75 GB RAM, and SSD based storage. The VMs were running Ubuntu 14.04 LTS with Linux kernel v.3.13. C3 Large instances provide enhanced networking capabilities via support for SR-IOV, which was enabled (as described in in [67]). Finally, the clocks on all VMs were synchronized using NTP.

We performed the same setup as described in Section 6.3.1.1. Two VMs were used to setup Cassandra's cluster and one VM was used to generate workload using the YCSB benchmark. (see Figure 6.6). All VMs were located in the EC2 datacenter in Frankfurt in a single availability zone. The RTT latency between each VM is roughly 0.5 ms. As in the local scenario, the Cassandra cluster was configured with the `replication_factor` of 2 (full data replication over R1

and R2). Additionally, `read_repair_chance` and `speculative_retry` have been turned off to exclude any additional communication between replicas. The YCSB VM was configured to generate low, medium, and high system loads using 8, 64, and 256 client threads. Each workload was configured for read only operations with the consistency level of TWO, such that upon R1 (R2) receiving a read request from YCSB, it generates an additional query to R2 (R1) to confirm the read value. Meanwhile, R1 (R2) waits for R2 (R1) to reply before sending a final response to YCSB. We repeated each experiment 10 times. YCSB was not a bottleneck for workload generation.

In this experiment, we use Tectonic to track and timestamp Cassandra's requests and responses exchanged between R1 and R2 in response to the workload generated by YCSB. Tectonic was configured to ignore client-to-server traffic, therefore messages produced in response to YCSB were not timestamped in this measurement (i.e., Tectonic only monitored the control traffic generated between R1 and R2).

The target of these experiments is twofold: first, we measure the length of time the application's requests spend traveling up and down the network stack and decouple this from the request service time and network latency. Second, we identify the dependence of each processing stage on the system's load.

6.3.2.2 User and Kernel Space Timestamping in EC2

Using the EC2 setup outlined in Section 6.3.2.1 and Tectonic's instrumentation we measure the time it takes for Cassandra's requests to be propagated from the NIC through the Linux network stack, then the time required for the application to produce a response, and finally the time it takes for the response to travel from the user space to the network driver.

Figure 6.8 contains 6 CDF plots arranged in 2 columns. The left column corresponds to the set of measurements obtained on EC2 and the right column illustrates results obtained from testing on bare metal installation, previously discussed in Section 6.3.1.2. Each plot contains 3 pairs of CDFs demonstrating: (1) the amount of time it takes for requests to travel up the network stack (red lines), (2) Cassandra's service time (blue lines), and (3) the amount of time it takes for the corresponding replies to travel down the network stack (green lines). These timings correspond to intervals t_3-t_4, t_4-t_5, and t_5-t_6 respectively in Figure 6.2. The solid and dashed lines represent measurements obtained on R1 and R2 respectively. Each column illustrate the correlation between the load on the machine (R1 and R2) and the amount of time it takes for a request to travel through each processing stage.

In all cases, the time spent in the upstream kernel and the service time demonstrate correlation with the load. In the case of EC2 Figure 6.8e show that

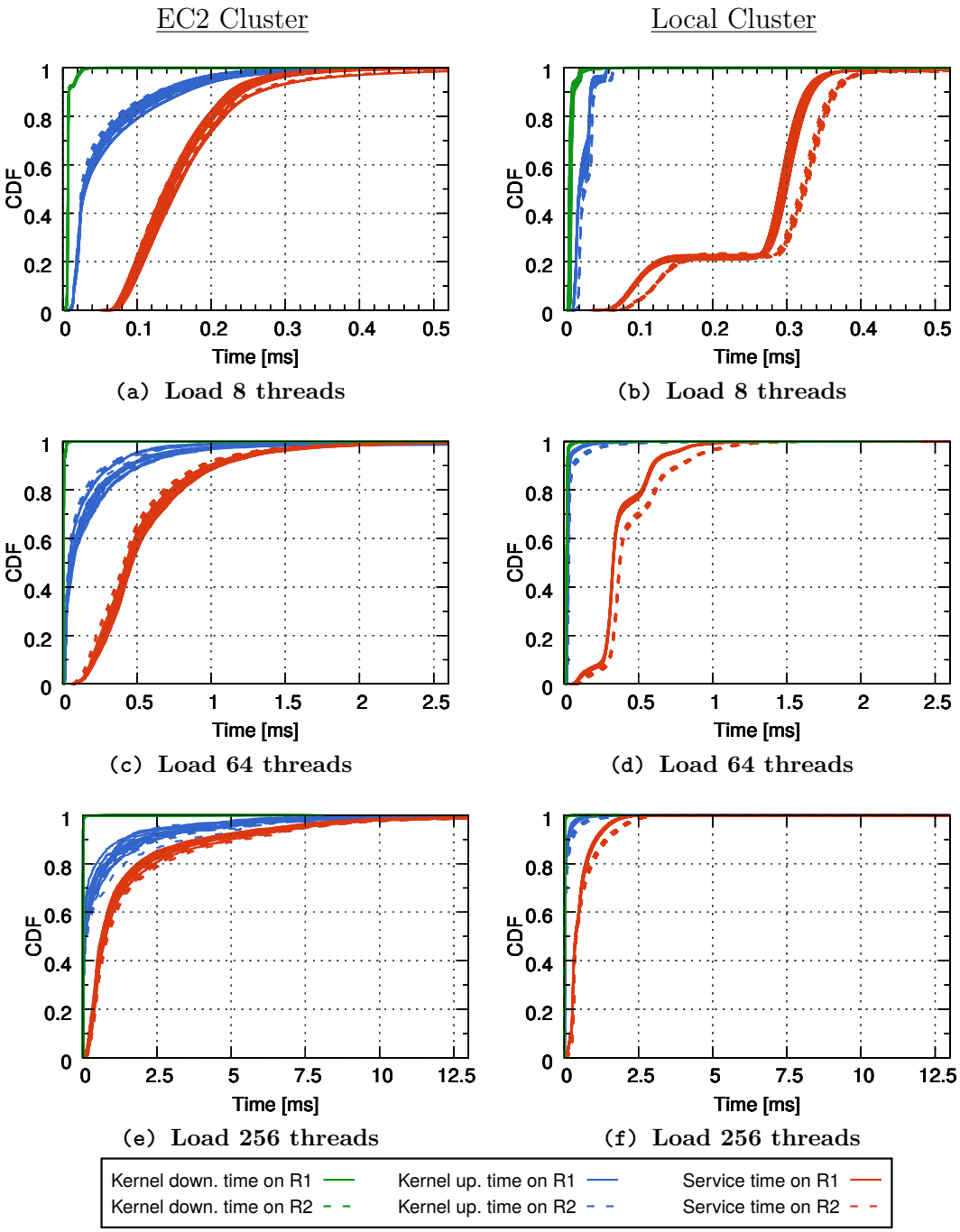
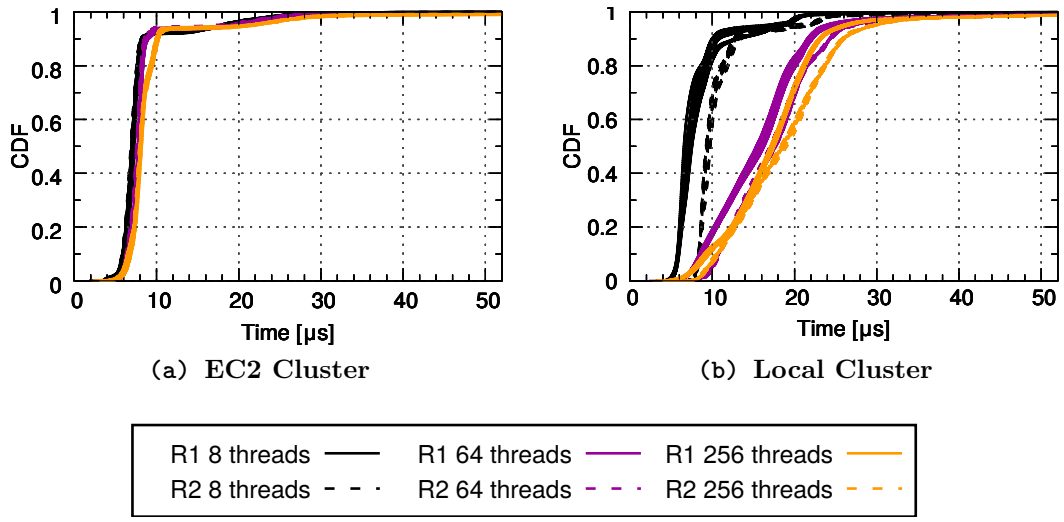


Figure 6.8: The amount of time spent by Cassandra's requests in the upstream, downstream TCP stacks, and the service time. Measurements correspond to the time intervals between t_3-t_4, t_4-t_5, and t_5-t_6 shown in Figure 6.2, on R1 and R2. Each experiment was repeated 10 times.



(a) EC2 Cluster (b) Local Cluster

Figure 6.9: The time it takes for a response to travel from the user space through the network stack until it reaches the `NF_INET_POST_ROUTING` hook in the kernel. Measured under 3 workload scenarios. This figure is the close up view of the downstream kernel time shown in Figure 6.8. The time intervals corresponds to t_5-t_6 in Figure 6.2.

under high load, kernel propagation time affects roughly 10% of the requests by more than 2.5 ms. However, further investigation showed that this is due to the application not being able to read data from the socket in time. Therefore, packets are queued in the TCP stack.

In contrast, bare metal installation demonstrate much weaker correlation with the load and more predictable performance. This could be due to load being too light for bare metal installation. Moreover, these measurements demonstrate performance differences between R1 and R2. The true cause for these deviations is not clear yet, as these machines have identical hardware and software configurations.

Downstream kernel time. Figure 6.9 shows the time it takes for a response to travel from the user space on R2 through the network stack until it reaches the `NF_INET_POST_ROUTING` hook in the kernel. In the case of EC2 (Figure 6.9), the downstream kernel time demonstrates near constant delay. The difference in the median propagation time under three load scenarios is under 1 microsecond. In contrast, while the bare metal installation demonstrates comparable performance under the light load, the downstream kernel time is more affected by the load of the machine. Both the median and the 99$^{\text{th}}$ percentiles inflated by approximately 15 and 25 μs respectively. While such deviation can be considered insignificant for a geo-distributed system, an additional measurements are required in order

to identify the true cause of such behavior. However, in the case of the EC2 deployment, these measurements suggest that the downstream kernel time can be taken as a constant and does not require a continuous run-time measurement.

6.4 EC2 Scheduler

The principal concept underlying cloud computing is hardware virtualization which allows a single hardware element (such as CPU, NIC, HDD/SSD, etc.) to be shared among multiple VMs (see Section 2.1 starting on page 9). This sharing is done in a seamless way by granting exclusive access to a given hardware component to each VM for a short period of the time or by having hardware that presents multiple logical instances of itself (as some NICs do). The order in which VMs access the hardware and the period of time that they can utilize this hardware are determined by the hypervisor's scheduling algorithm. Intuitively, such a scheduling algorithm has a profound impact on the individual VM's performance and the overall hardware utilization. Consequently, the algorithm has to efficiently make scheduling decisions, while being fair in providing the designated share of resources to each VM.

In this section, we seek to measure and understand virtualization delays associated with the sharing of computing resources (specifically CPUs) by analyzing Amazon's EC2 public cloud. In Section 6.4.1 we give an overview of the default XEN scheduler [68]. In Section 6.4.2 we describe our method and the set of measurement probes that we deployed on EC2 VMs to infer the scheduling policies utilized. Next, in Section 6.4.3 we outline different types of EC2 instances and group them based on their CPU sharing scheme and then describe the setup for our evaluation. Finally, in Section 6.4.4 we present our findings.

6.4.1 XEN Credit1 scheduler

Amazon EC2 uses a customized version of the XEN hypervisor to deploy a variety of VMs. This hypervisor is responsible for the hardware resource allocation among deployed VMs. Each XEN VM contains one or more Virtual CPUs (VCPUs), forming a set of VCPUs running on a single physical machine. For each VCPU present in a VM, the scheduling algorithm of the hypervisor determines the order in which each VCPU is allocated to a Physical CPU (PCPU). A single PCPU can execute instructions of only one VCPU at a time. Therefore, when more than one VCPU is scheduled on a PCPU, this creates delays as each VCPU has to wait for its turn to execute. The waiting time depends upon the number of competing VCPUs and the scheduling quanta, i.e., the unit of time given to each VCPU to execute. The length of the scheduling quanta is selected based upon a trade-off between

processing intensive and latency sensitive tasks. Processing intensive operations benefit from longer periods of uninterrupted execution (which improves their CPU cache efficiency), while latency sensitive tasks (such as the majority of interactive services relying on cloud implementations) want to achieve fast response times and therefore benefit from shorter, but more frequent scheduled allocations. Note: reducing the size of the scheduling quanta leads to increased overhead, as it forces the PCPUs to spend a larger proportion of their time computing the schedule and there is a penalty when switching between VCPUs, thus reducing the effective utilization of the hardware, specifically the PCPU (and its associated caches). The default scheduling quanta used in XEN is 30 ms, but values as low as 1 ms are also considered to be acceptable [68].

Today, the default scheduling algorithm in XEN is the Credit Scheduler [68]. This algorithm is based on crediting and debiting of CPU credits among currently active and scheduled VCPUs. In this algorithm, each VCPU receives CPU credits distributed at a constant rate. The credit allocation rate is proportional to the VM's *weight* - an administratively defined property, allowing certain VMs to consume greater/lesser fractions of computing resources. When a VCPU is running on a PCPU, its credits are debited in proportion to its running time.

In a credit scheduler, a VCPU can be in one of four states: (1) VCPUs that exceeded all of their CPU credit are assigned to the OVER state, (2) VCPUs that have a positive amount of CPU credit remaining are in the UNDER state, (3) VCPUs that are idling or waiting for I/O assigned are in the BLOCKED state, and (4) VCPUs that were awakened from the BLOCKED state (for example, by receiving an I/O interrupt) are in the BOOST state. A PCPU maintains a list of scheduled VCPUs in a priority queue subdivided into three priority ranges (associated with OVER, UNDER, and BOOST) in accordance with the possible states of VCPUs**.

A currently running VCPU can be preempted in three cases: (1) when it runs out of credit, (2) when it becomes idle (i.e., waiting for an I/O), and (3) when a new VCPU enters the BOOST state. In the latter case, the currently running VCPU is given a timeslice before its preemption (with a suggested configuration ranging between the default of 30 ms and as small as 100 us). When a non-idle VCPU is preempted from the active state, it is placed at the end of the priority range according to its state (i.e., the end of the UNDER or OVER ranges). The next VCPU to be moved into the active state is picked up from the BOOST state. If no VCPUs are present in BOOST state then one from UNDER and subsequently OVER states are picked instead.

Note, the XEN scheduler employs a system-wide load balancing policy, such

**BLOCKED VCPUs are excluded from the priority queue until they wake up, in which case they are moved into the BOOST priority range.

that if a PCPU does not have VCPUs in its local queue in the UNDER state, it will look for VCPUs in the UNDER state from other PCPUs *before* it will process VCPUs from the OVER state in its local queue. To control the migration of VCPUs among PCPUs, the XEN scheduler provides two configuration options: *soft* and *hard CPU affinities*. *Soft affinity* defines a set of PCPUs where a VCPU *prefers* to run. For example, this property can be set on a multi-socket Non-Uniform Memory Access (NUMA) machines, to discourage VCPU migration between different CPU sockets. In contrast, *hard affinity* defines a set of PCPUs where a VCPU is *allowed* to run. This facilitates flexible resource allocation by limiting VCPUs migrations and providing dedicated PCPUs to individual VMs.

XEN's Credit scheduler operates in terms of static time intervals - called quanta (with a default value of 30 ms). Every quanta worth of time a new scheduling decision is made and potentially a new VCPU is selected in a round robin fashion to be moved into the active state. Every 10 ms the Credit scheduler distributes new credits among non-active VCPUs and checks if the active VCPU is idling.

Moreover, XEN's Credit scheduler supports two operational modes: *work-conserving* and *non work-conserving*. In the work-conserving mode, a VCPU can execute on a PCPU for a virtually unlimited amount of time, if no other VCPU is scheduled. In contrast, in the non work-conserving mode, each VCPU cannot exceed a capped amount of the PCPU time, even if the PCPU is not busy running other VCPUs. For more detail descriptions of XEN's Credit schedule, please refer to [69, 70]

6.4.2 Methodology

In this section, we attempt to gain insights into EC2's scheduling algorithm by observing time intervals of when a VCPU is running on a PCPU and the time intervals when the VCPU is inactive (i.e., waiting in one of the priority queues).

In this measurement, we employ techniques similar to those used in Cyclictest [71] a popular benchmark for real-time Linux Kernels. On the VM under test, we deploy our Interval Measurement Probe (IMP) (see the source code in Appendix A.3). IMP is a user space process that attempts to measure equally spaced time intervals. The probe contains a loop which calls `clock_gettime(CLOCK_MONOTONIC)` to obtain a nanosecond precision monotonic timestamp and subsequently sleeps for a short interval of 10 μs using s `usleep()` call. Note, the actual duration of the sleep may be slightly over 10 μs due to system activity, the time it takes to process the call and by the granularity of the system timer. However, we expect this value to be relatively small such that inter loop intervals will not be erroneously perceived for the intervals of the VM's VCPU preemption. During each iteration of the loop, the time difference between the current timestamp and the timestamp obtained in the previous iteration of the

loop (time delta) is computed and preserved for future analysis. The time delta between two consecutive iterations of the loop includes the time it takes to execute the set of instructions comprising the loop logic, the sleeping time of 10 μs and the time it takes for the OS to wake up the IMP process and to perform the context switch.

A prolonged time interval between two consecutive timestamps (above 1 ms) indicates that the IMP process has not been executed on this PCPU for the given period (in this text, we refer to such intervals as a Preemption Intervals (PIs). Such long inter loop intervals can be caused by one of two reasons: **first**, the IMP process was not scheduled for execution due to the VM's Linux scheduler giving execution priority to another kernel or user space process(es) that might have been using this PCPU during that time. **Second**, the VM itself was not scheduled on the PCPU by the XEN hypervisor. Thus, none of the VM's processes were running at that time. In our measurement, we are interested in the latter cases.

To reduce the possibility of the IMP process not being scheduled on the VM we changed the status of the process to be a Real Time process and changed the default Linux scheduler to SCHED_FIFO. SCHED_FIFO is a real-time Linux scheduling policy that implements a first-in first-out scheduling algorithm. SCHED_FIFO does not have a minimum scheduling quanta and the real-time process currently executing can only be preempted in three cases: (1) by other real-time process of a higher priority, (2) when the process is blocked performing I/O, or (3) when the process intentionally finishes running (i.e., sleeping or calling an `exit`). The priority of the IMP process has been set to 99, the maximum possible value for real-time processes. No other real-time processes were being executed by this VM. As a result, this process should have always be able to execute and should have executed without preemption.

A VCPU of a VM can be preempted from the PCPU either when the VCPU runs out of credit, another high priority process enters the BOOST state, and when the VCPU become idle. It is impossible to directly affect the first two cases, as EC2 does not provides visibility into the number of competing VCPUs. However, to prevent the VM in this test, from being placed into a BLOCKED state by the XEN hypervisor we must ensure that the VM never sleeps, does not perform I/O, and consumes all the available CPU cycles. To address these three issues, we implemented an artificial workload generator via an infinite loop running on the same VM. The workload was executed as a user space process with the lowest possible priority of +19 set using the `nice` system call. The workload process is operating under the default Linux scheduling policy SCHED_OTHER. This policy has lower priority with SCHED_FIFO. The difference in the processes priorities and the two different scheduling policies guarantees that the IMP process would never be preempted by the workload process, while at the same time, IMP would

preempt the workload process as soon as it wakes up from its (micro) sleep. The number of workload processes and IMP probes was equivalent to the number of CPU cores available to the VM. Each pair of processes was pinned to a single core using the `taskset` command.

We performed 2 sets of measurements: using the wall clock `gettimeofday()` and the monotonically increasing time `clock_gettime(CLOCK_MONOTONIC)` The `gettimeofday()` provides the wall clock with microsecond precision. However, this clock is also affected by the changes in the system time, for example, caused by NTP adjustments which operates at a millisecond precision. In contrast, the `clock_gettime(CLOCK_MONOTONIC)` provides relative time (an arbitrary moment determined at the last system boot time) with nanoseconds precision. Unless stated otherwise, the rest of this section presents results based on the monotonic clock measurements.

6.4.3 Evaluation Setup

Amazon EC2 provides a wide range of VM instances, each designed to target certain customers' demands. However, in terms of CPU scheduling all EC2 instances can be divided into 3 categories: *Burstable* performance, *Fixed* performance, and *Dedicated* instances.

6.4.3.1 Burstable Performance Instances

Burstable performance instances are designed for non-continuous CPU usage. The idea behind these instances is similar to the Credit scheduler used by XEN. Each instance continuously receives CPU credit in accordance with its size. The available CPU credit is shared among all VCPUs of a given instance. When a VCPU is running on a PCPU the corresponding amount of credit is debited. Idling VMs accumulate unspent CPU credit up to a maximum of 1 day worth of CPU credits.

Table 6.1 shows the list of burstable instances tested in this work. For example, a `t2.nano` instance receives 3 CPU credits per hour. Each credit is equivalent to 1 minute of a PCPU time. If the instance uses less than 3 CPU credits per hour, it accumulates the excess (with a limit of 72 CPU credits). The value, 72 CPU credits, would allow a `t2.nano` instance to consume 100% of a PCPU time for 72 minutes without being preempted, given that there is no other competing VCPU.

Table 6.1: **Evaluated Burstable performance, general purpose EC2 instances.**

Instance Name	vCPUs	Initial CPU credit	CPU credit/hour	Base CPU utilization (%)	CPU credit cap
t2.nano	1	30	3	5	72
t2.micro	1	30	6	10	144
t2.small	1	30	12	20	288
t2.large	2	60	36	60	864

6.4.3.2 Fixed Performance Instances

Fixed performance instances are designed to provide stable performance guarantees. The performance of these instances measured in compute units - Amazon's custom definition of units of computing resources. Performance of one compute unit is approximately equivalent to the performance of a 1.0-1.2GHz Xeon processor from the year 2007. unlike Burstable instances, fixed performance instances do not accumulate CPU credit. Moreover, these instances are subject to hardware sharing. Table 6.2 shows a list of Fixed performance instances tested in this section.

Table 6.2: **Evaluated Fixed performance EC2 instances.**

Instance Name	vCPUs	Compute Units	Description
m3.medium	1	3	General Purpose
m3.large	2	6.5	General Purpose
c3.large	2	7	Compute Optimized
c3.xlarge	4	16	Compute Optimized

6.4.3.3 Dedicated Instances

Dedicated instances are deployed on dedicated hardware and are isolated from other customers. Such deployment is designed to provide the most stable performance, hence the performance should be equivalent to bare metal hardware performance. In this work, we only evaluated burstable and fixed performance instances.

6.4.4 Evaluation

This section outlines the experiments that we performed on EC2 using the burstable and fixed performance instances enumerated in Tables 6.1 and 6.2.

Unless stated otherwise, each experiment was performed for 24 hours. Using `cron`, we scheduled IMPs to be run every 20 minutes for 19 minutes on each

VM participating in an experiment. Each run of IMP produced (1) a histogram containing frequencies of the inter loop time deltas of length under 1 ms with nanosecond precision, thus each histogram contains 1M records, and (2) a list of time deltas of length over 1 ms containing a timestamp and the length of the preemption interval. A one-minute interval between two consecutive runs of the IMP was reserved for the I/O operations needed to flush the histogram and the timestamps log to disk. Over the course of 24 hours, each VM performing the measurement accumulated $24*(1hour/20min) = 72$ pairs of files per VCPU. Based on these log files we compute the following statics: **(a)** aggregated preemption intervals and aggregated active times over 1 ms (based on all of the log files per VM), **(b)** a number of preemptions per 19 minute sampling period, and **(c)** a distribution of inter loop intervals (< 1 ms) on a single log basis.

The range of EC2 instances used in this measurement span between 1 and 4 VCPUs, all running Ubuntu 14.04 LTS with Linux kernel v.3.14. To produce a performance reference point we repeated our measurements on a physical machine at our local premises. Our local setup was equipped with a dual socket 8-core Intel Xeon CPU E5-2667 v3 operating at 3.20 GHz. Hyper-threading was turned off and the OS is the Ubuntu 14.04 LTS distribution with Linux kernel v.3.14.

The rest of this evaluation is organized as follows. In Section 6.4.4.1 we evaluate burstable EC2 instances in two scenarios: with and without available CPU credit. In Section 6.4.4.2 we measure preemption intervals of the Fixed performance instances and compare them to measurements made on the bare metal hardware installation at our local premises. After this, in Section 6.4.4.3, we look at the distribution of the sub-millisecond intervals measured by IMP and the minimum time delta achieved by each instance type.

6.4.4.1 Evaluating Burstable Instances

In this section, we evaluate scheduling intervals of the burstable EC2 instances (as listed in Table 6.1). We performed two sets of experiments: **first** we evaluated the case when VMs under test did *not* have any excessive CPU credit. The EC2 API was used to validate the amount of available CPU credit per instance (Figure 6.10 left column). **Next**, we repeated the same measurements using the same type of EC2 instances with an excess of CPU credit. To accumulate an *excess* of CPU credit per instance, all instances were left idling for 24 hours before the start of the measurement. In contrast to the no-credit evaluation, this evaluation lasted between a minimum of 79 minutes for `t2.nano`[††] samples and 10 hours for

[††]The `t2.nano` instances have maximum CPU credit cup of 72 minutes and accumulate an additional 3 credits per hour. Three IMP cycles per hour consume $19 * 3 = 57$ minutes, leaving an excess of 3 minutes. Therefore, the total CPU credit adds up to approximately 79 minutes, allowing us to execute the IMP measurement 4 times.

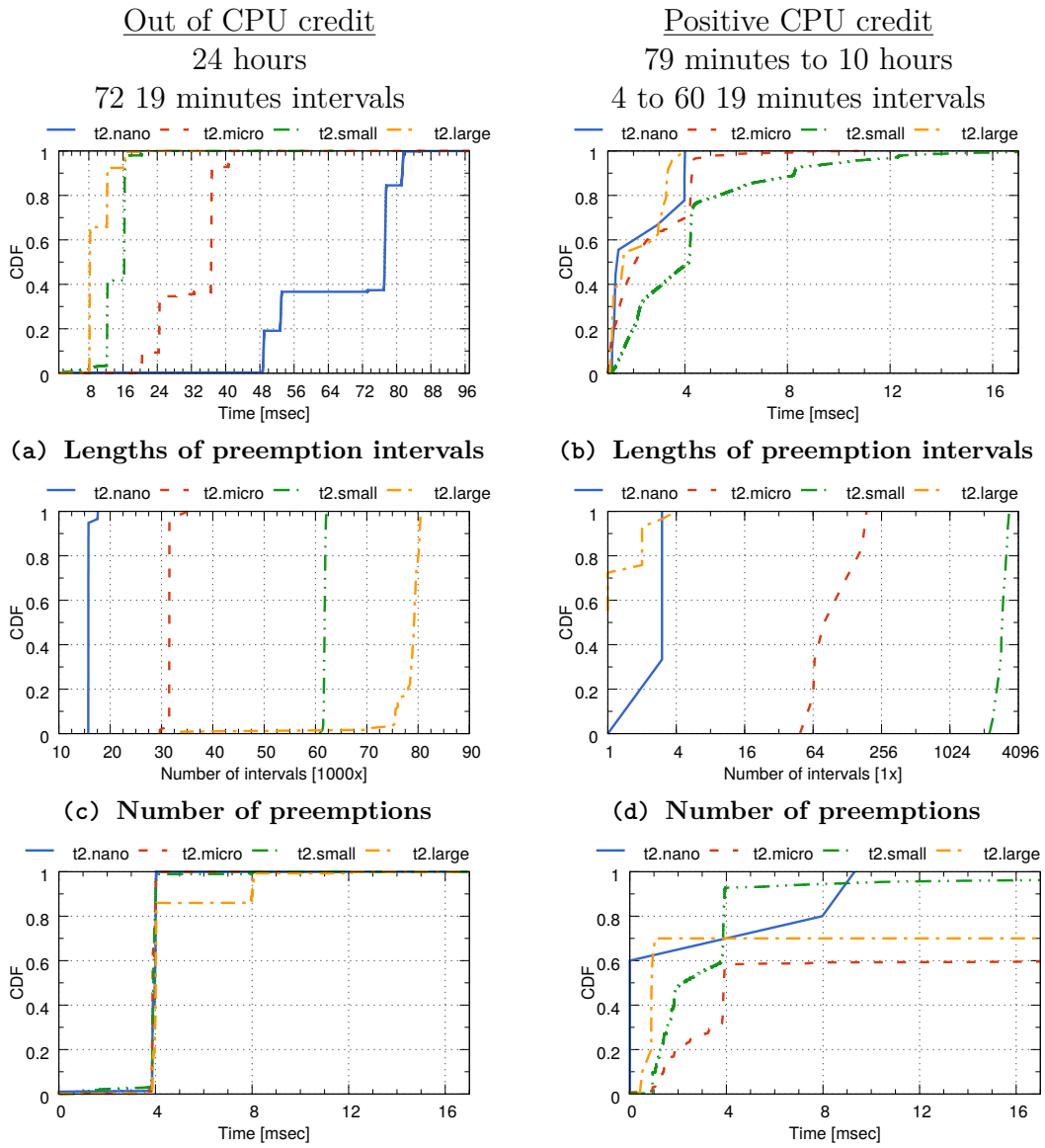
`t2.large` until these VMs used up all of their CPU credit allocation (Figure 6.10 right column).

The left column in Figure 6.10 illustrates our measurements for the no-credit case. Figure 6.10a shows the aggregate lengths of PIs (i.e., the time intervals when the IMP process was not running on PCPU). This figure shows that the PIs are multiples of 4 ms for all instances and *negatively* correlated with the CPU credit allocation per VM type. For example, the `t2.large` instance has the shortest PIs, while the `t2.nano` has the longest with 20% of all PIs lasting over 80 ms. Figure 6.10c shows the CDF of the aggregate number of PIs over all IMP measurements performed on the given VM (i.e., 72 and 144 of the all 19 minutes intervals for single and dual core instances). The number of PIs is correlated with the CPU credit allocation per EC2 instance. However, Figure 6.10e shows that all instance types get an equal time quanta per allocation. This measurement shows that all types of `t2` instances were given an equivalent short timeslice of the PCPU time (mostly 4 ms at a time); however, instances with a lower credit allocation spent more time in the scheduler's priority queue, waiting to be moved into the active state. The above observations suggest that the XEN scheduler operates in a non-work conserving way for T2 instances that run out of CPU credit.

Next, we look at the case when the evaluated VMs had an excess of CPU credit (Figure 6.10 right column). Figure 6.10d shows that 40% of IMP measurements on `t2.large` instances experienced between 1 and 4 preemptions. Similarly, the smallest instance `t2.nano` experienced only 1 to 3 preemptions per 19 minutes. In contrast, `t2.micro` and `t2.small` experienced 100s and 1000s of PIs per 19 minutes respectively.

These measurements demonstrate that the excess in CPU credit in T2 instances does not guarantee non-preemption for a prolonged period. Moreover, the initial size of an instance (i.e., its CPU quota per hour) does not affect PCPU allocation when an excess of CPU credit is present.

Finally, Figures 6.10b and 6.10f illustrate that less than 50% of preemption intervals and less than 50% of the active intervals are multiples of 4 ms. In the XEN Credit scheduler such a condition could arise when a competing and currently active VCPU goes into the BLOCKED state *before* consuming its allocated quanta of $4X$ ms (where X is an integer). Next, the VCPU of our VM can be moved from the UNDER state to the ACTIVE state by the scheduler, thus creating an interval that is not a multiple of 4 ms. Scheduling behavior when there is an excess of CPU credit corresponds to the work conserving mode (note: in the case of no CPU credit we operate in the non-work conserving mode).



(a) Lengths of preemption intervals

(b) Lengths of preemption intervals

(c) Number of preemptions

(d) Number of preemptions

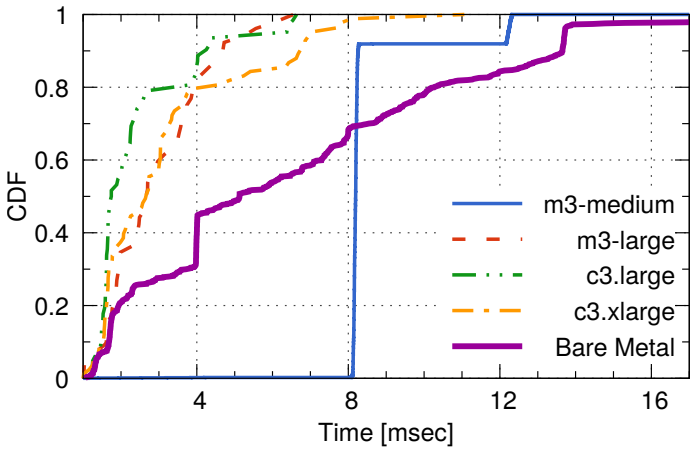
(e) Lengths of active intervals

(f) Lengths of active intervals

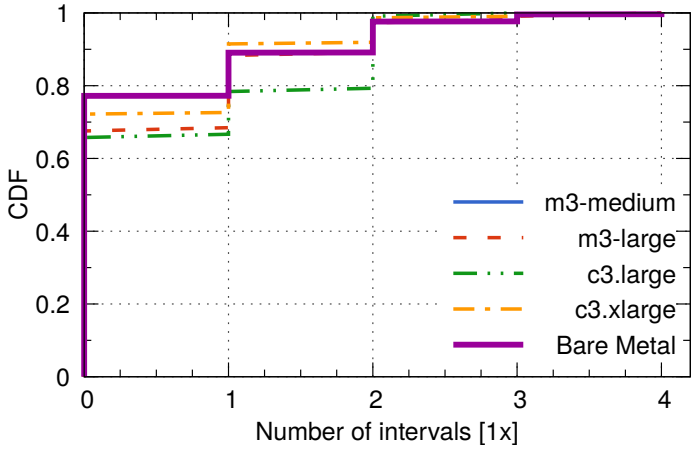
Figure 6.10: Top row demonstrates the aggregate CDFs of the lengths of preemption intervals (i.e., when IMP process was not running). The middle row shows the CDFs of the number of preemptions (above 1 ms) per IMP sampling interval (i.e., per 19 minutes). The bottom row shows the aggregate CDFs of the lengths of active intervals. The three plots in the left column shows results obtained from EC2 instances with no spare CPU credit available, while the right column corresponds to the case where each instance has a positive CPU credit.

6.4.4.2 Evaluating Fixed Performance Instances

Next, we repeated our measurements using Fixed performance instances (for those instances listed in Table 6.2) located in one availability zone of a single EC2 datacenter. All instances were placed as close as possible in attempt to detect correlated behaviors. For these experiments we choose the EC2 datacenter in Oregon.



(a) Lengths of preemption intervals



(b) Number of preemptions

Figure 6.11: The lengths and the number of IPs measured for Fixed performance EC2 instances. The solid purple line shows the control test performed on the bare metal Linux server.

Figure 6.11b shows that 70% of IMP measurements had no preemption intervals above 1 ms. Out of remaining 30% of IMP samples 80% of PIs were under 4 ms

(see Figure 6.11a). The control test using bare metal setup demonstrates the same probability of encountering PIs over 1 ms, this suggests that these PIs where likely to have been caused by the Linux scheduler, rather than the XEN hypervisor scheduler. While `m3.medium` belongs to the same subclass of EC2 instances as `m3.large`, it demonstrates a scheduling policy similar to that of the `t2` instances discussed in Section 6.4.4. The number of PIs measured per sample for `m3.medium` is above 60K and has not been shown in Figure 6.11b as this would have obscured the differences in the number of preemptions for the other types of instances.

Surprisingly, Figure 6.11a illustrates that the bare metal setup exhibits longer delays than were measured in Amazon's public cloud. That 20% of all PIs in the control test were above 12 ms could be caused by the fact that control test was running 16 threads. Moreover, the control test also demonstrates a long tail of 329 ms (not shown in the figure), this is likely to be caused by a local I/O operation potentially caused by NFS.

Note, the `c3` and `m3` instances (except for `m3.medium`) demonstrate very stable performance comparable to the bare metal installations. PIs do *not* show a strong correlation with multiples of 4 ms.

The results observed in these measurements depend upon several factors. It is possible that the machines where we were running VMs for these measurements did not experience heavy interference from co-located instances, hence the total number of VCPUs were less than or equal to the number of PCPUs present in the hardware. A longer measurement campaign could show if the observed good performance is the norm.

6.4.4.3 Sub-Millisecond Intervals

In this section, we examine the distribution of time intervals between two consecutive iterations of the IMP loop that are smaller than 1 ms and could not be caused by PCPU scheduling at the hypervisor level. In these measurements, we removed the artificial workload processes and the 10 μs sleep call in each iteration of the IMP process (see Section 6.4.2). Thus, each iteration of the IMP was executed as fast as possible on a given VM.

Figure 6.12 shows sub-millisecond intervals measured on four `t2.micro` VMs located in different datacenters of EC2. Each curve corresponds to a histogram measured over 19 minutes by an IMP. All four sub-figures demonstrate similar shapes with visible spikes around 2, 4, and 12 μs. This observation suggests that tested VMs were deployed on identical hardware, which is common for a single cloud provider. The nature of these spikes is likely to be affected by the PCPU frequency and hardware clock frequency of the hardware where these VMs were deployed.

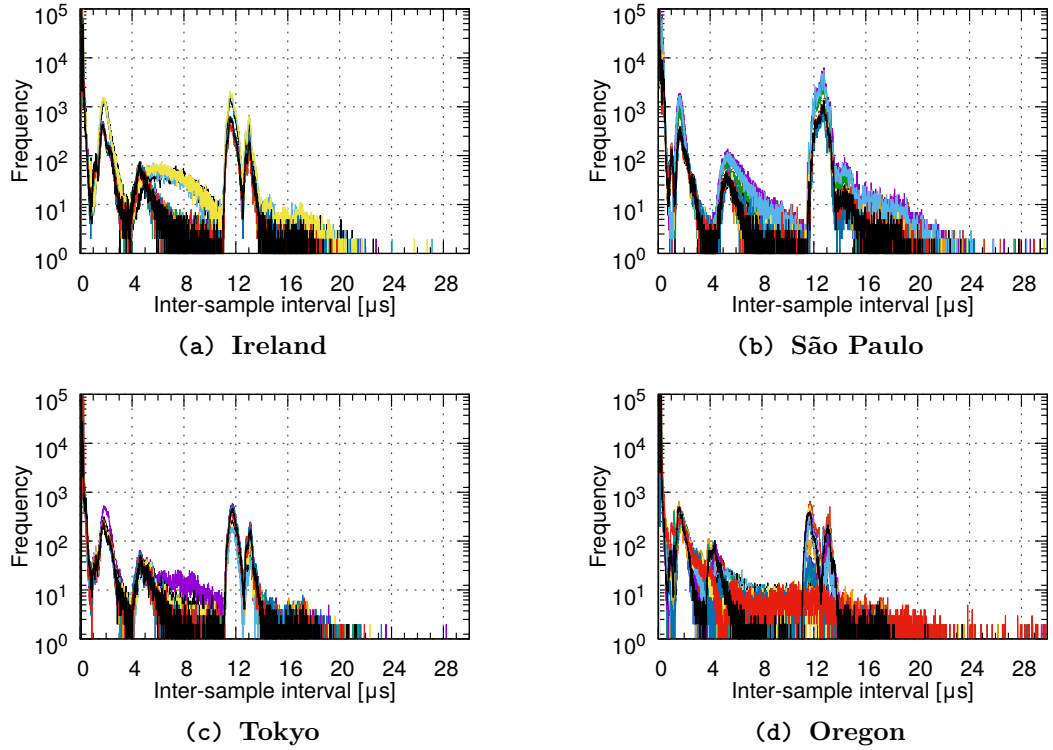


Figure 6.12: Inter-loop intervals measured in four t2.micro instances over 1 day. Each color corresponds to a particular 19 minute sampling interval.

Figures 6.12a, 6.12c, and 6.12d show a subset of samples (curves) that produce different probability distributions of interval lengths. The shapes of these figures have different peak levels of between 4 and 8 μs. The change in distribution over time could be caused by external interference (from co-located VMs) or due to the migration of the VM to different hardware.

Next, we repeat our measurements using 6 `c3.large` EC2 instances located in a different datacenters (California, Ireland, Oregon, São Paulo, Tokyo, Virginia). Figure 6.13 shows a similarity in shapes among all 6 instances. However, the interval distribution is different between `c3.large` and `t2.micro` instances, suggesting that they are deployed on a different hardware.

Figures 6.13b, 6.13c, and 6.13d demonstrate consistent distribution intervals across two sets of measurements (as the shape of the distribution stays the same). However, Figures 6.13a and 6.13e show a large difference in the delay distribution in some of its samples. Moreover, note that the peak in Figure 6.13a around 12 ms is similar to the spikes observed in Figure 6.12 measured in T2 instances.

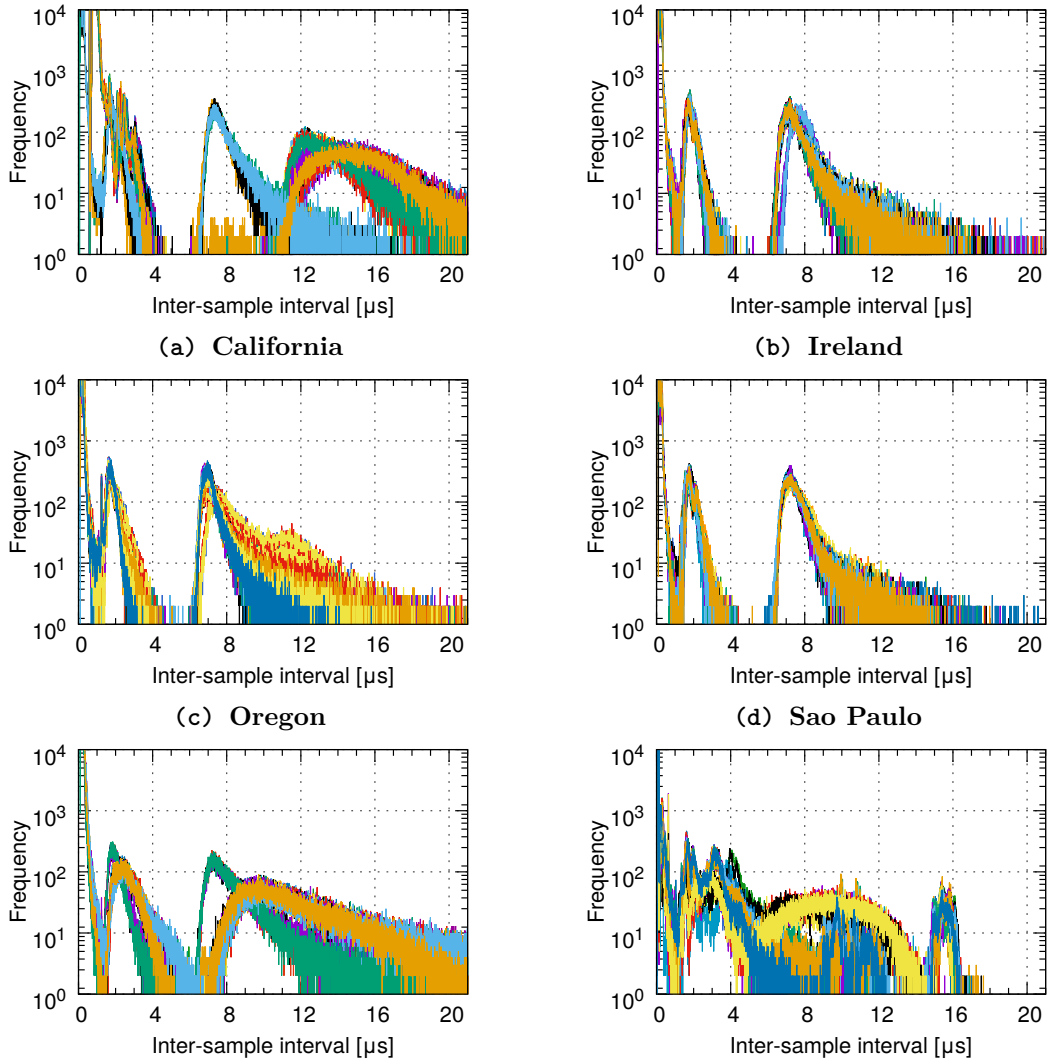


Figure 6.13: Distribution of the time intervals between two consecutive iterations of IMP loop. Each color corresponds to a particular 19 minute sampling interval. Subfigures (a) to (e) show results for C3.large instances in 6 different datacenters. Figure (f) is the control measurement obtained at the local bare metal installation.

6.5 EC2 Scheduler Inferring Via Tectonic

In this section, we used Tectonic's to infer hypervisor scheduling policy, without running an explicit IMP process. We used a similar methodology as described in Section 6.4.2, however, instead of measuring equally spaced time intervals between consecutive iterations of the loop, we measured intervals between consecutive

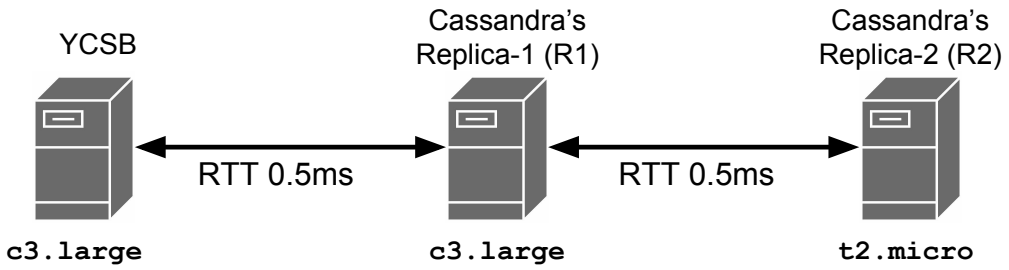


Figure 6.14: Three VMs collocated in one availability zone in EC2 datacenter in Frankfurt. Two VMs (R1 and R2) comprise Cassandra cluster. The third VM (on the left) generates workload using the YCSB benchmark. The workload is directed only to R1.

timestamps produced by Tectonic as a response to incoming (outgoing) application messages. We refer to this metric as Inter-Sample Intervals (ISIs).

The experiment setup (shown in Figure 6.14) is identical to our previous EC2 measurements described in Section 6.3.2.1, except for the following changes. **First**, for fine-grained control over the experiment, we modified YCSB to allow requests to be sent only to R1. Thus, when R1 receives a read request, it generates an additional query to R2 and waits for R2 to reply before sending a final response to YCSB. **Second**, the YCSB and R1 were deployed on `c3.large` instances while R2 was deployed on `t2.micro`. This has been done in order to isolate the impact of VCPU preemptions to R2 only. Moreover, `t2.micro` instances have only a single VCPU which simplified initial measurements. **Finally**, before each measurement we drained R2 of all of its CPU credit. This has been done by running a sample workload (not part of the measurement) until R2 run out of CPU credit. Using EC2 API we measured the amount of CPU credit available to R2.

6.5.1 Methodology

In this section, we describe the methodology that we used to interpret measurement results obtained from Tectonic. A VCPU can be in the active state (i.e., currently being scheduled on a PCPU) or in preempted state (i.e., waiting to be moved into the active state). The amount of time a VCPU spends in each of these states is typically multiple of 4 ms (see Section 6.4.4.1).

Figure 6.15 demonstrates a schematic representation of transitions between active and preempted states, experienced by a VCPU as a result of the hypervisor CPU scheduling. The X axis represents time and gray circles represent application messages passing through Tectonic's timestamping location (this could be either one of 8 possible timestamping locations see Figure 6.2). The moment when

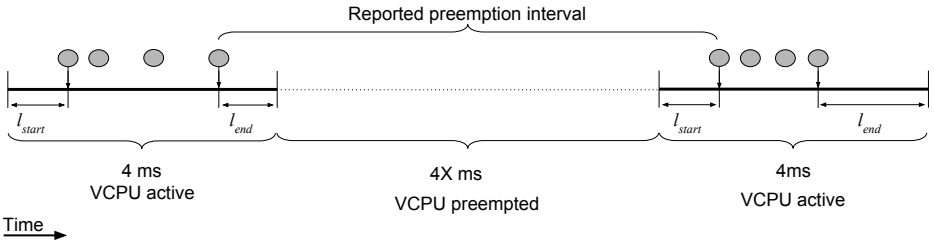


Figure 6.15: Two intervals of 4 ms when the VCPU was active, separated by an interval of VCPU preemption (in the middle). The length of PI is multiple of 4 ms. Gray circles correspond to packets passed through Tectonic and indicate the moment when a timestamp was generated.

the first packet passes through Tectonic and generates a timestamp does not perfectly align with the beginning of the VCPU's active interval. This is due to several reasons. For example, a kernel or a user process needs to be scheduled by the VM's Linux scheduler to move application messages forward through the processing pipeline; hence it might take some time until the first packet will reach a timestamping point.

In Figure 6.15, we denote l_{start} as the time delay from the moment when the VCPU becomes active to the moment when Tectonic timestamps the first packet. Similarly, we denote l_{end} as the time interval from the moment when the last packet was timestamped by Tectonic to the moment when the VCPU got preempted by the hypervisor.

The distribution of timestamps within an active interval depends on the application workload, the number of outstanding requests (packets), and the VM's Linux scheduler. Naturally, the ISI between any two timestamps within an active interval cannot exceed the duration of that interval (i.e., 4 ms)[‡‡]. Thus, all ISIs below 4 ms correspond to a single active interval, while ISIs above 4 ms include the length of the PI. The ability to identify periods and lengths of PIs is critical for inferring hypervisor scheduling policies, hence we look into it more closely.

The reported lengths of ISIs above 4 ms are affected by the lengths of l_{end} and l_{start} (see Figure 6.15). The minimum length of the l_{start} is bounded by the minimum time it takes for a VM to schedule and execute the process responsible for packet (message) propagation and the time required to perform timestamping, hence the minimum length of l_{start} is non-zero. Moreover, the l_{start} is likely to be longer for Tectonic's timestamping in the user space. In contrast, the length of l_{end} is dependent only on the moment when the VCPU got preempted by the hypervisor. Potentially, a VCPU can be preempted right after the moment when

[‡‡]According to our previous measurements the expected length of active intervals for a `t2.micro` instance is 4 ms (see Figure 6.10e).

Tectonic generated a timestamp, thus producing l_{end} of a near zero length.

Since l_{start} is non zero, we can conclude that reported lengths of *all* ISIs of length above 4 ms are overestimated. Moreover, in the case when the sum of l_{end} and l_{start} exceeds the length of the scheduling quanta (e.g., above 4 ms) the length of the PI can be overestimated by the length of the scheduling quanta[§§].

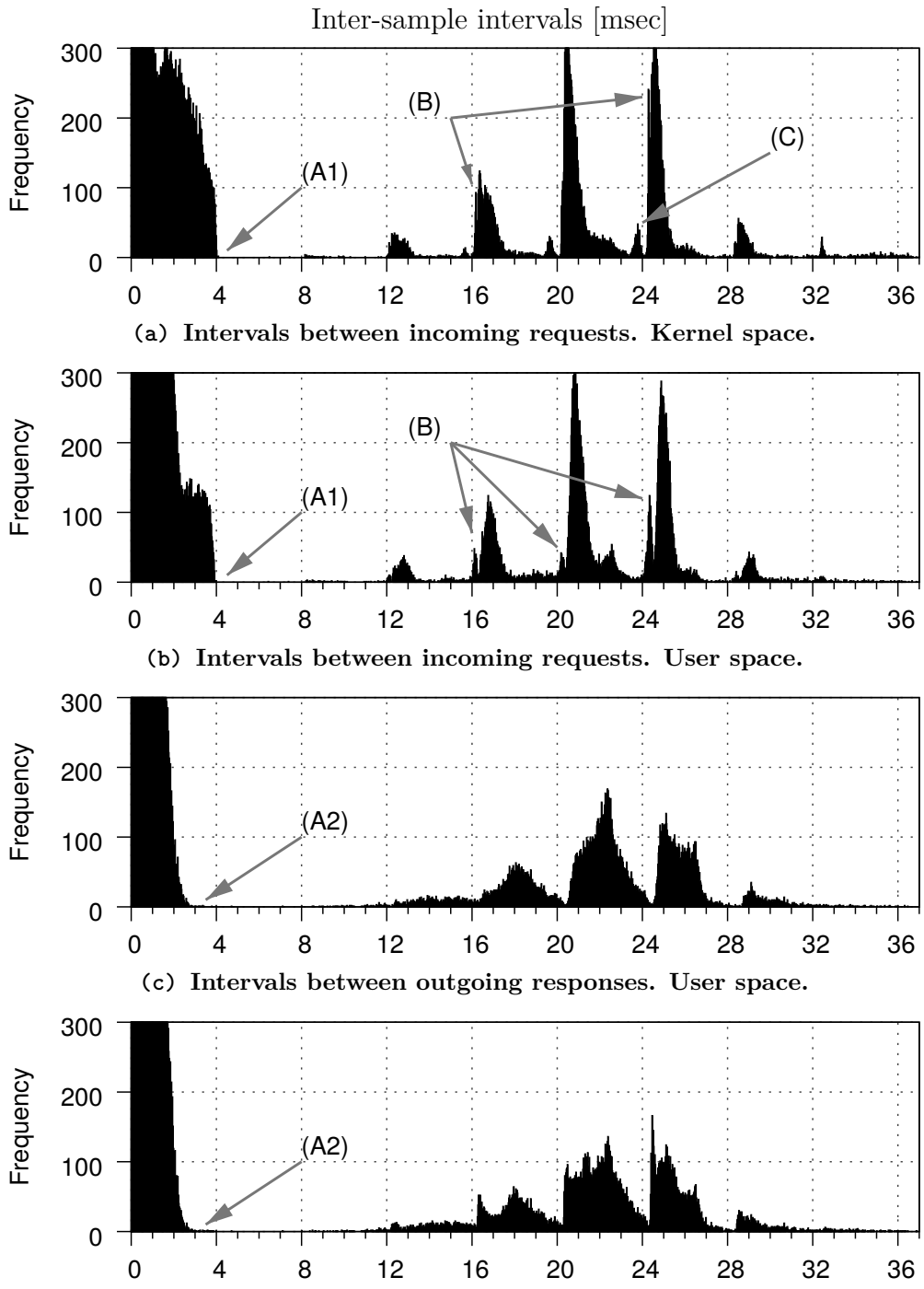
The length of the l_{start} depends on the state of the R2 at the moment when it was preempted and the external states of YCSB, R1, and the hypervisor at the moment when the VM became active. The state of the VM before the moment of preemption defines the locations of unprocessed packets (messages) in the processing pipeline (e.g., messages stuck in the TCP networking stack). Therefore, when R2 transition into the active state, two distinct cases are possible. The **first** case occurs when outstanding packets (messages) exist in the processing pipeline *before* the timestamping location (i.e., Tectonic's timestamping locations in the kernel and user spaces). For example, if the NIC has unprocessed packets in RX queue (this could happen if the VM was preempted before it could finish processing outstanding packets or if new packets have arrived while R2 was preempted), then l_{start} will depend on the VM's Linux scheduler to start processing the packets. Thus, the time when the first packet will be timestamped is likely to be short and result in a sub-millisecond length of the l_{start}.

The **second** case occurs when R2 does not have any packets in the processing pipeline at the moment when it becomes active. Therefore, the length of l_{start} for a given timestamping location will depend on the other replica (R1) to send requests that will generate timestamps on R2 upon arrival. In the example of our topology (see Figure 6.14), if R2 is being preempted for 10s of milliseconds, then it is likely that YCSB and R1 will be blocked and waiting for R2 to reply before they can send new requests. Therefore, when R2 transition into the active state, it will take at least 1 ms (network RTT between R2 and YCSB) for the new requests to start arriving at R2 and for Tectonic to start generating timestamps. Hence, the expected length of l_{start}, in this scenario, is above 1 ms.

6.5.2 Evaluation

We begin by examining the distribution of ISIs recorded on R2 by Tectonic. Figure 6.16 illustrates measurements obtained under the load of 64 YCSB client threads. Each plot is a histogram of ISI timestamped at $\texttt{t}_3$, $\texttt{t}_4$, $\texttt{t}_5$, and $\texttt{t}_6$ on R2 (see Figure 6.2). These figures show how often ISIs of a particular length were encountered on R2 during our measurements. All histograms clearly demonstrate distinct ISIs bands that correspond to PIs multiple of 4 ms. A set

[§§]To mitigate such overestimation it is possible to combine Tectonic's timestamps collected at different locations (user and kernel spaces) in a single VM.



(a) Intervals between incoming requests. Kernel space.

(b) Intervals between incoming requests. User space.

(c) Intervals between outgoing responses. User space.

(d) Intervals between outgoing responses. Kernel space.

Figure 6.16: Histograms of ISIs recorded by Tectonic at t_3, t_4, t_5, and t_6 on R2.

of labels highlights interesting aspects of this measurement. Next, we discuss our preliminary evaluation of these results.

Labels **(A1)** in Figures 6.16a and 6.16b show a clear cut in ISIs at 4 ms. This observation is orthogonal to our measurements of the hypervisor's scheduling policies used in burstable EC2 instances (previously discussed in Section 6.4.4.1). In that section, Figure 6.10e on page 114 demonstrated that the prevalent number of active intervals in `t2.micro` instances has a length of 4 ms. Moreover, Figure 6.10e illustrated that the PIs for a `t2.micro` instance are over 12 ms. Therefore, histograms of ISIs in Figure 6.16 match our previous observations. The clear cuts indicated by labels **(A1)** illustrate a distribution of ISIs within a single scheduled quanta of 4 ms.

The fact that ISIs within a scheduling interval can span almost the entire length of the scheduling quanta indicates high burstiness in request processing (i.e., two bursts of packets timestamped at the beginning and the end of the active interval respectively).

In contrast, labels **(A2)** in Figures 6.16c and 6.16d do *not* demonstrate a clear cut of ISIs and diminish at the maximum length of roughly 3 ms. Two factors can contribute towards these observations. First, application requests spent the longest period of time in Cassandra while being serviced (see Figure 6.8), this increases the length of l_{start} before the first response is being timestamped. Second, when a burst of requests being processed by Cassandra, the time between the first and the last response is likely to be spread wider than the burst of packets traveling through the TCP network stack. This reduces the maximum ISI length observable within a single period of activity.

Labels **(B)** in Figures 6.16a and 6.16b show small elevations near the start of the active intervals. These elevations correspond to set of PIs measured with a high accuracy where the sum of l_{end} and l_{start} adds up to a fraction of a millisecond. This case can occur when timestamping the remaining unprocessed requests at the time when R2 became active (i.e, the remaining packets in the RX queue and upstream TCP networking stack). Note, the user space histogram in Figure 6.16b demonstrates a higher frequency of such events. This is due to unprocessed packets in RX queue also being timestamped by Tectonic at the user space.

Finally, label **(C)** in Figure 6.16a illustrates a frequent occurrence of ISIs at the border range of the scheduling quanta. We inclined to believe that this is also an indication of the burstiness in our setup and can be explained by periodic occurrences of PI with a long (near the length of the scheduling quanta) l_{start} and a short l_{end}. For example, this can happen if R2 processes a burst of outstanding packets from the RX queue and getting preempted before it can process the new requests. Meanwhile, new requests arrive at the NIC. Upon the next active interval, R2 processes new set of packets. The sequence of these events will result in a ISI

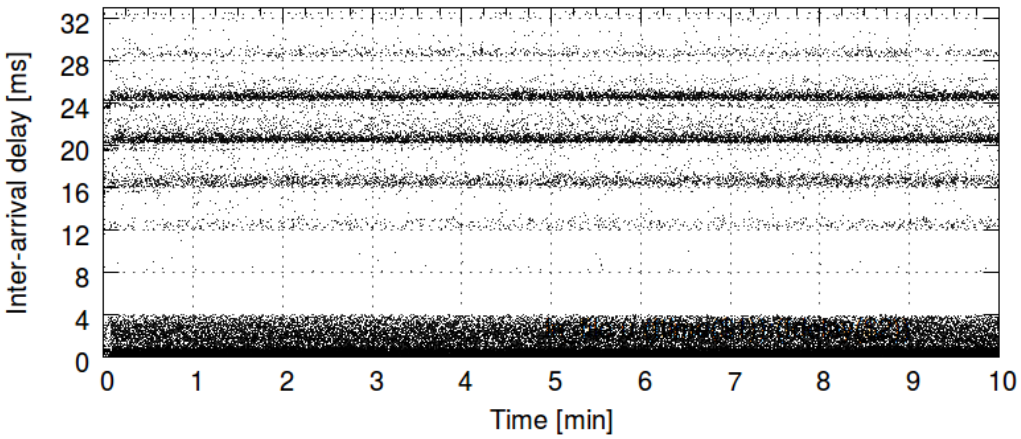


Figure 6.17: Requests' ISIs recorded in the kernel space in R2. This figure corresponds to the histogram shown in Figure 6.16a. Each point represents an ISI. The Y axis indicates the length of the interval. The X axis indicates time since the start of the experiment.

which length is overestimated almost by the length of the scheduling quanta.

Figure 6.17 demonstrates requests' ISIs recorded in the kernel space on R2 and corresponds to the histogram shown in Figure 6.16a. This figure illustrates that the distribution of lengths of PIs throughout the measurement is stable. While PIs of 20 and 24 ms appear more frequently then PIs of different length, the remaining challenge is to predict the length of upcoming PI at run-time.

6.5.2.1 Conclusion and Future Work

In this section, we demonstrate how Tectonic's timestamping can be used to infer the hypervisor's scheduling policy and PIs on a VM. The aim of this measurement is to derive a method of using scheduling information pro-actively at run-time. For example, if a replica is aware of the current scheduling intervals of other replicas, it can incorporate the lengths of the expected PIs in its decision making process and hence, make better decisions. This can be done by adjusting the perceived network latency to a remote replica by the expected delay associated with the CPU scheduling of the remote VM.

However, the following questions remain to be answered. While R2 can measure and estimate it's own scheduling policy using Tectonic, an additional analysis is required to validate that R1 can also infer this information by monitoring ISIs of replies (requests) received (sent) from (to) R2. Moreover, in the case when R1 is also deployed on a burstable EC2 instance this will produce an additional challenge in estimating the scheduling policy of R2 as two scheduling policies (on R1 and

R2) are likely to interfere with the measurement. Finally, an additional analysis that evaluates the effects of the load (i.e., the number of YCSB client threads) on the distribution of PIs is necessary.

6.6 Summary

In this chapter, we introduced our ongoing work in describing an end-to-end request completion time as a function of network latency and the load on the remote machine. We described the architecture and the scope of use of Tectonic a tool that performs user space and kernel space timestamping of the targeted application's messages. By using Cassandra as an example, we demonstrated how Tectonic could be used to measure propagation times of the application's requests and responses.

The obtained measurements showed that the amount of time requests spent in upstream TCP networking stack and during the service time is the function of the load on the machine. In contrast, the downstream TCP networking stack introduces near constant delay despite the load on the machine.

To better understand the deployment conditions in public clouds we studied the effects of virtualization on request propagation. Next, we showed how Tectonic's timestamps could be used to infer XEN's CPU scheduling policies. We discussed how this information could be utilized by distributed systems at run-time while making replica selection decisions. We leave the implementation of this functionality for our future work.

Chapter 7

Related Work

The performance of geo-distributed systems depends on a wide range of factors including networking, cloud computing and virtualization, software verification, and various domain specific algorithms. Naturally, each of these areas by itself is a substantial research area. Therefore, the following sections simply summarize research contributions in those areas related to this specific licentiate thesis.

7.1 Symbolic Execution

Symbolic execution has a long history, and some of the first works are [72, 73]. EXE [43], KLEE [44], JPF-SE [74], CUTE, and jCUTE [75] are modern examples of symbolic execution tools. Most of these tools are being updated by academia and open source communities.

Symbolic execution is not limited to discovering bugs in applications. For example, Siegel [76] uses the SPIN [77] model checker and symbolic execution techniques to compare and verify the correctness of the Message Passing parallel implementation of an algorithm given its sequential counterpart. Person [78] proposes differential symbolic execution to determine differences between two versions of the same application. KleeNet [79] applies symbolic execution to a network of wireless sensors. It runs on unmodified applications, while automatically injecting non-deterministic events common in distributed systems.

In contrast with this earlier work, GeoPerf does not require non deterministic events or event reorderings; the replica selection algorithms are explored in a completely deterministic environment. This dramatically reduces the number of possible code paths to explore, and therefore reduces the search time requirements. We apply symbolic execution to a new problem, and overcame the difficulties that arose. In particular, our focus has been on performance-related (specifically latency) issues.

7.2 Replica Selection

Geo-distributed storage systems tend to forward clients' requests towards the "closest" replicas to minimize network delay and to provide the best performance. This is a common task e.g., in self-organizing overlays [80]. One of the primary tasks is to correctly compute or estimate the round trip delay among the nodes. Various systems have tackled this problem. Vivaldi [81] piggy-backs probes on application messages in order to infer virtual network coordinates. GNP [82] performs measurements from dedicated vantage points to compute network coordinates. Alternative approaches for estimating network distance (and hence infer or determine the RTT) include [62, 83, 61].

The next step is to use the estimated delay to select an appropriate replica to contact. Meridian [84] is a decentralized, lightweight overlay network that can estimate the delay to a node in the network by performing a set of pings that are spaced logarithmically from the target (i.e., each consequent ping is sent from an overlay node that is closer to the destination). OASIS [85] is a system built on top of Meridian, but additionally incorporates server load information. DONAR [86] also argues for a decentralized approach, by using a set of mapping nodes, each running a distributed selection algorithm to determine and assign the closest replica for each client. C3 [87] reduces long tail latency via careful replica selection by introducing feedback concerning the load on the servers and a distributed rate control is used to avoid sending additional requests to already overloaded servers. In our work, we do not argue that we have the best replica selection algorithm, but rather provide a tool that can find flaws in the performance of such algorithms.

7.3 Cloud Computing

Cloud computing provides a common platform for deploying and running systems that are geo-distributed across the globe. While cloud platform allows simple deployment of applications, it remains up to the application itself to determine which geographic region or regions to choose for replication and which network transport protocols to utilize. Volley [88] addresses the problem of automatic data migration based on a multitude of factors, such as client activity, replication policy, and datacenter capacities.

The cloud providers differ in the number of datacenters and their geographic locations, services availability, and cost of usage (typically based on numerous metrics of network and hardware utilization). Ang Li, et al.[22] compared four popular cloud providers by measuring the performance of their inter and intra datacenter networks, storage services, and overall costs. They observed significant differences among the four providers that they tested. The logical continuation of

this work is to deploy on-line services on top of multiple cloud providers, in order to exploit the best factors of each provider. For example Wu, et al.[17] developed SPANStore, a key-value store that can span across multiple cloud providers, while striving to satisfy latency, fault tolerance, and consistency constraints, and minimizing overall cost of deployment. Bessani, et al. [24] evaluated the possibility of using multiple clouds in order to increase the security of the data being stored.

However, to date, it is still uncommon to utilize multiple cloud providers, primarily because of the differences in cloud interfaces, hardware configurations, and billing costs. As a result a finite set of datacenters of a single cloud provider acts as an ecosystem for the deployed application, thus placing a finite limit on the variety of deployment and replication policies. For example, Amazon EC2 currently has 10 geographic locations, each with an average of 3 availability zones. This limited number of datacenters significantly reduces the search space for satisfactory dissemination strategies.

7.4 Network Measurements

Existing work in the area of latency measurements can be generalized into two categories. The first category [32, 31, 30, 35, 34] concentrates on studying the impact of virtualization on the network and application performance in a cloud environment. These works show that sharing hardware resources can have a negative effect on latency, throughput, and bandwidth of applications running on these virtual machines. Moreover, these performance artifacts are very different from those that occur in non-virtualized environments and these artifacts are often hard to predict.

The second group of works [89, 51, 90] study packet loss, delay, and jitter over geo-distributed WAN network paths. An interesting comparison of cloud providers is done in CloudCmp [22], where the authors computed the CDF for optimal paths from 260 clients provided by the PlanetLab testbed [91] to the datacenters of four cloud providers; three datacenters from Amazon were tested among other configurations. They report that the latencies among datacenters are highly dependent upon their geographic locations. Latencies between different cloud providers are often incomparable due to the different geographic locations of their datacenters. These results suggest that any static configuration of the replica selection algorithms should be always tailored to deployment on a specific cloud provider.

However, to the best of our knowledge, our study is the first to show the correlation of geo-distributed network paths among all regions of one cloud provider, and to demonstrate how such dynamics can affect the selection of the nearest replica.

7.5 Stream Sampling

Data sampling has important applications in numerous areas: medicine, economy, computer systems and networking. The general problem can be summarized as how to select a subset of samples from a potentially infinite stream of data such that this small subset would be representative of the initial data. This is often done for the reasons of optimization and performance, particularly in cases of Big Data where storing the entire set of samples in memory is infeasible or the resulting computational time makes it impractical.

Research in this area has a long history. Vitter [92] proposed several methods for sequential random sampling from a pool of samples with a known size. Later he removed the requirement for knowing the initial size of the pool in his follow up work [93].

Aggarwal et al. [94] highlights the advantages of a biased rather than unbiased methods of reservoir sampling; then he proposed new biased sampling algorithms and defined their useful properties, such as the maximum size of the reservoir.

Cormode et al. [46] proposed a novel approach to use a forward decay function in biased reservoir sampling. This technique uses the relative time from the start of sampling rather than the absolute time of an individual sample's arrival, when determining the probability of inserting a given sample into a reservoir. This method was later used as a basis of Cassandra's [6] Dynamic Snitch replica selection algorithm.

While there is a lot of work in deriving various sampling algorithms that fit a specific purpose, little or no work has been done to answer the question of when a particular reservoir has changed sufficiently to be re-evaluated. This is particularly important for data streaming applications, such as replica selection algorithms.

7.6 Transport Protocol Optimizations

Winstein et al. [95] have demonstrated that RemyCC is capable of generating congestion control algorithms based on a traffic model and objectives, such as high throughput and low queuing delay. In their evaluation, an automatically generated congestion control system was able to outperform existing human-designed techniques including TCP Vegas [96], NewReno [97], and Cubic [98]. This demonstrates that prior knowledge of network conditions can improve the performance of the generated algorithm. Moreover, no single implementation is optimal for all types of networks and application requirements. Sivaraman, et al. [99], further explored automatically generated protocols. While current results seem to be very promising, it remains unclear whether it is possible to

automatically generate a protocol that outperforms a human-designed counterpart under a variety of network conditions and topologies.

Existing TCP protocols rely on the slow start mechanism, where a congestion window is incremented (usually exponentially) and smoothed RTT is measured over the course of several segment exchanges. Utilizing EdgeVar's knowledge of network paths and latency classes, we can immediately provide the relevant latency information, thus accelerating the TCP slow start mechanism.

Today, TCP is widely used in cloud datacenters (Alizadeh et al. [100] reports 99.91% of traffic in a Microsoft datacenter was TCP); however, default congestion control mechanisms fail to provide high link utilization and low latency, while facilitating a wide range of flow sizes. To address this problem, DCTCP [100] relies on Explicit Congestion Notification (ECN) notifications by switches to report the amount of network congestion. Unlike other congestion control mechanisms that reduce the size of the congestion window by half, upon detecting congestion, DCTCP reduces the window's size in proportion to the fraction of packets that are marked by ECN. Thus, this allows the protocol to maintain stable link utilization in the presence of competing traffic.

DX [101] proposes a high precision (sub-microsecond) latency based congestion feedback mechanism. Using either software (DPDK) or hardware (NIC) based packet timestamping, one-way delay measurements, and additional packet queues DX removes various sources of variability in the RTT measurement due to network stack, packet batching, and NIC queuing. As a result, Lee et al. [101] achieved queuing of only a few packets while achieving 10 Gbps throughput.

Moreover, TCP was never designed to operate in a virtualized environment. Delays associated with virtual CPUs of VMs not scheduled on a physical CPU can erroneously suggest the presence of network congestion, thus increasing the number of retransmissions and increase latency, while simultaneously reducing link utilization. PVTCP [102] proposed a technique to detect periods of VCPU preemption using the rate of change in Linux system clock (also known as `jiffies`). When a VM is executing its system clock is incremented regularly by a periodic interrupts. Each interrupt increases `jiffies` by one. However, after the period of preemption, VM's system time is updated by the hypervisor, thus the value of `jiffies` is incremented proportionally to the duration of preemption. Irregular long jumps in Linux system time indicates preemption of the VM. If a period of preemption is detected, then all outstanding TCP RTO timers are adjusted by the duration of the preemption (to avoid triggering retransmission of packets that arrived when the VM was not scheduled) and RTT measurements received in this period are ignored. In contrast to our approach, PVTCP does not address delays associated with the VM scheduling on the receiver and simply treats these delays as a network delay.

Vigfusson et al. [103] outline the limitations of IP Multicast: its lack of reliability, flow control, and limited scalability in terms of the number of multicast groups used. They propose MCMD, an application level protocol, that addresses the scalability limitation by multiplexing a limited number of hardware multicast groups at the routers, while remaining completely transparent to the application that is using it. Chu et al. [104] outlined the same protocol limitations and developed Narada, an application level implementation of IP Multicast. This work demonstrates the possibility of constructing a dissemination network on top of the dynamic and heterogeneous Internet environment.

Petlund et al. [105] study the suitability of SCTP for latency sensitive thin streams. They report that despite SCTP often being used as a transport protocol for thin-traffic signaling, its latency performance (based on its default configuration) is inferior to TCP. They addressed this issue by modifying timers and retransmission policies of SCTP, but also argue that it is important to apply new configurations in the presence of only thin-traffic streams.

When designing a network transport protocol and modeling its performance, it is hard to cover all possible deployment scenarios, and this becomes even harder as networks evolve and become more complex. Many authors agree; for example Sivaraman, et al. [99] argue that a single protocol configuration *cannot* fit all network states and system needs. Custom protocol configuration adjustments driven by dynamic real time conditions have a great potential for future networks.

7.7 Timestamping

Arlos and Fiedler [106] evaluated timestamping accuracy of various hardware and software timestamping techniques. In their experiments, they generated a stream of temporally equally spaced probes and timestamped their arrival at the destination host. By observing the inter-arrival times of packets and comparing them to the expected interval, they estimated the precision of various timestamping methods. Orthogonal to their work, Sommers et al., [107] developed MAD, a tool for coordinated scheduling of probe emission that allows high precision probe generation under high load and for multiple parallel experiments. Orosz and Tamas [108] demonstrated how libpcap can operate with a nanosecond precision using only software based timestamping.

Time Synchronization A great deal of work has been done in the area of time synchronization [109, 110, 111]. A comprehensive overview of methods and associated difficulties with performing OWD measurements can be found in [112, 113]. Luckie et al. [114] proposed Time Sequence Latency Probes (TSLP) to identify and locate the presence of congestion in inter-domain routing. Unlike other techniques, this method does not require instrumentation of both the source

and the destination. To identify the presence of congestion between two points along the path to the destination, TSLP sends probes to the neighboring and distant routers. Measurements are compared to previous observations; an increase in RTT to the distant router but not the neighboring router suggests the presence of congestion between these two routers.

Chapter 8

Conclusion

TODAY, it is hard to find an area of human lives that have not been affected by the modern technological advances in the form of online computing services. The performance and functionality demands for these services are constantly increasing, and this trend is likely to persist and grow in the future. The ongoing competition among service providers results in a persistent demand for the better performance needed for development of new and innovative services. To evolve alongside with the expected demands, distributed services are trying to be more adaptive, smarter, and quicker. Distributed systems that fail to follow along will become extinct and replaced by their competitors.

Throughout this thesis, we addressed a set of challenging technical problems standing in the way of geo-distributed systems today. Our main aim was to improve the quality of run-time decisions performed by these systems. Our work can be summarized in three steps: (1) understand the underlying deployment conditions experienced by geo-distributed systems in third party clouds; (2) validate the correctness of the core logic responsible for making run-time decisions, and (3) provide an accurate and timely input for the core logic, to guarantee its effectiveness.

Our work begins with an extensive cross datacenter latency measurement described in Chapter 3. In this work, we illustrated the high degree of variability in network conditions across WAN. Based on these observations, we argued for the importance of dynamic adaptations to changes in network conditions. However, the existing techniques, used by the industry for making run-time decisions, vary a lot and often change between consecutive releases. Important questions such as: (1) which network metrics to use, (2) how often to change your decision, and (3) *what is* the right decision are not at all evident.

To answer these questions, we designed and developed techniques for testing and validating the core logic responsible for making run-time choices. In Chapter 4 we introduced GeoPerf, a tool for systematic testing of replica selection algorithms.

GeoPerf uses symbolic execution and light-weight modeling to compare two algorithms in a controlled environment of the event based simulator. Using application's source code, GeoPerf generates specific latency inputs that expose weaknesses and performance anomalies in replica selection algorithms. We applied GeoPerf to test two popular storage systems (Cassandra and MongoDB) and found two bugs in their replica selection implementations.

Our next step was to provide an accurate and timely input for the systems that perform run-time decisions. First, we looked at the existing latency estimation techniques that are commonly used by geo-distributed systems deployed in the cloud. We found that these techniques have many limitations and do not provide an accurate view of the network states.

In Chapter 5 we introduced EdgeVar, a network path classification and latency estimation tool. This tool decomposes a noisy latency signal into two components: base latency associated with routing and residual latency variance associated with network congestion. Moreover, it uses active (via traceroute) and passive (FR) techniques to detect changes in routing and notifies the systems accordingly. EdgeVar provides explicit latency estimation and speeds up the process of adaptation by allowing geo-distributed systems to make a quick decision based on the accurate view of the network state.

In Chapter 6 we demonstrated that virtualization could introduce delays in the order of 10s of milliseconds. We showed how Tectonic could be used to infer hypervisor's scheduling policy, and proposed an approach of how this information could be used by the application during the run-time decision making process. The aim of this work is to combine the knowledge of network latency obtained from EdgeVar with the knowledge of service time to describe an end-to-end request completion time as a function of network latency and the load on the remote machine.

The combination of the individual steps discussed in this work yields a cleaner and more accurate perception of the network and system states. Consequently, having access to this information at run-time facilitates the decision making process in the unstable environment of geo-distributed deployments and provides a solid foundation for building distributed systems of the future.

8.1 Future Work

This section, outlines a few possible directions for our future work.

Conclude Work on Tectonic. First, we plan to conclude the remaining work on Tectonic, as currently stated in Section 6.5.2.1. Our aim is to finish the integration of Tectonic with EdgeVar and to describe an end-to-end request completion time as a sum of individual delays associated with network latency and service time.

Next, we seek to combine information theory techniques with Tectonic's (latency, OWD, traceroute, and FR) to determine the optimal sampling technique based on the current state. The benefit of this approach is twofold: first, we effectively reduce the overhead of using Tectonic by reducing the number of traceroute requests and the amount of network traffic being sampled, and second, this enables us to detect shifts in latency classes and react rapidly to routing changes.

Using Tectonic we want to incorporate our findings into the design of a novel replica selection module based on the clear latency representation, knowledge of the load, processing delays, and scheduling policies utilized across a geo-distributed cluster. Finally, we plan to incorporate this module into existing geo-distributed systems that can benefit from the clear view of the system and network states (e.g., Cassandra datastore) and perform extensive evaluation by deploying the modified system across EC2.

Analyzing Long Term Latency Samples. To date, we have collected over 1 year of latency samples across all geo-distributed datacenters of Amazon EC2. The vast array of collected data can be used for many evaluations that we did not perform due to time limitations. For example, the data can be used to evaluate the temporal change in the QoS across EC2. This will show us how network latency changed over time, particularly when new datacenters were added in new geographic locations. This data set provides many opportunities for future research.

Second Order Reservoir Sampling. In this thesis, we were regularly coming back to the important question of the correct and timely latency estimation. An accurate perception of the network state is the foundation for the higher level application logic, allowing distributed systems to make better decisions when necessary.

Today, common latency estimation techniques, collect run-time measurements and preserve individual latency samples in a reservoir of a finite size (see Section 5.4.1). The subset of latency samples in a reservoir is used to estimate statistical properties (such as median or the 99th percentile) of the underlying latency distribution. However, the evaluation of these statistical properties is performed at discrete time intervals, which leads to a stale representation of the input stream in the intervals between recomputations. In Section 4.4.3 we briefly looked into the problem of periodic sampling reservoir resets and the consequent effects of this action.

The static nature of the reservoir recomputation intervals does not work well within dynamically changing conditions of the Internet. If the recomputation interval is too long, a system can miss potentially critical events reflected in the intermittent state of the reservoir. Conversely, if the interval is too short, this

will lead to a waste of CPU cycles, increase of energy consumption and decrease scalability of a system. Both cases lead to suboptimal system performance.

In the case of Cassandra, the inability of the system to dynamically adjust recomputation interval leads to the "worst case scenario" configuration, designed to react to high frequency changes in network conditions. For example, Cassandra recomputes latency sampling reservoirs every 100 ms for each replica. Moreover, in the current implementation of Cassandra, each reservoir recomputation is synchronized with the higher level replica selection logic and this logic is triggered regardless of whether there is a change in the state of the reservoir. Doing so, wastes additional resources.

The main question that we intend to answer is: *how to dynamically adjust the reservoir recomputation interval at run-time, while insuring the ability of the application to react to sudden changes in network conditions?* Our current approach is based on monitoring the arrival of new samples that are being added to the reservoir and use this information as an indication of a change in a target percentile. For example, by knowing the current value of the median percentile of a reservoir, we can track the number of new samples being added on the left-hand side and right-hand side of the current median value. Consequently, we can look at the ratio of these samples (the ratio is expected to fluctuate around 1/2, if there is no change in the actual latency distribution), while the sudden change of rate can indicate a need for reservoir recomputation.

Our preliminary results indicate that this technique can outperform algorithms based on static reservoir recomputation intervals, by increasing the precision and reducing the total number of recomputations. However, a more thorough study is required.

Predicting Long Tail Events. Unfortunately periodic run-time latency sampling cannot detect sudden changes in the tail of the latency distribution. The long tail events are infrequent by the definition and it might take multiple samples to realize a change in the tail, at which point the ongoing requests are likely to incur an extended delay. Existing proactive techniques that address the long tail latency experienced over WAN, are based on the redundancy which increases network and system load, as additional requests needs to be sent and processed.

In our future work we seek to answer the question: *how to predict the arrival of infrequent long tail latency samples?* Our preliminary approach relies on the higher order statistical analysis to match the current state of the reservoir with the historically observed states, on a given network path, that indicate the possibility of long tail events occurring. The variable latency conditions observed on EC2, during our measurements, will act as a training dataset.

Using Principal Component Analysis (PCA) we will identify significant statistical metrics that can help in reservoir classification. For example, a state

of a reservoir can be represented as a multidimensional vector containing a subset of target percentiles and their corresponding rates of change. The output of this analysis will indicate a likelihood of tail events to occur and can be further use along with other tail reduction techniques. For example, we can rely on redundant requests only when the probability of them to occur is high enough.

Our latency decomposition technique presented in Chapter 5 significantly helps with both tasks: (1) deciding on when to recompute the reservoir and (2) predicting the occurrence of long tail events. By removing routing changes from the latency stream, we isolate the data present in the reservoir to be associated with the congestion and the amount of competing traffic on the network. Thus, working with the residual latency signal is expected to improve the precision of our technique.

Bibliography

[1] K. Bogdanov, M. Peón-Quirós, G. Q. Maguire Jr, and D. Kostić, "The nearest replica can be farther than you think," in *Proceedings of the Sixth ACM Symposium on Cloud Computing*. ACM, 2015, pp. 16–29.

[2] "Jammr," https://jammr.net/ Accessed 30-Nov-2015.

[3] J. Marescaux, J. Leroy, M. Gagner, F. Rubino, D. Mutter, M. Vix, S. E. Butner, and M. K. Smith, "Transatlantic robot-assisted telesurgery," *Nature*, vol. 413, no. 6854, pp. 379–380, 2001.

[4] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, p. 8, 2013.

[5] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems (TOCS)*, vol. 16, no. 2, pp. 133–169, 1998.

[6] Apache, "Cassandra," http://cassandra.apache.org/ Accessed 30-Nov-2015.

[7] "Mongodb," http://www.mongodb.org/ Accessed 30-Nov-2015.

[8] Amazon, "Aws acceptable use policy," http://cassandra.apache.org/ Accessed 01-Sep-2016.

[9] ——, "Aws customer agreement," https://aws.amazon.com/agreement/ Accessed 01-Sep-2016.

[10] W. (Boston) and MIT, "1963 timesharing: A solution to computer bottlenecks," https://www.youtube.com/watch?v=Q07PhW5sCEk Accessed 9-Jul-2016.

[11] "Amazon ec2," http://aws.amazon.com/ec2/ Accessed 30-Nov-2015.

[12] "Google cloud," https://cloud.google.com/ Accessed 30-Nov-2015.

[13] "Microsoft azure," https://azure.microsoft.com Accessed 30-Nov-2015.

[14] "Ibm cloud," https://www.ibm.com/cloud-computing/ Accessed 09-Jun-2016.

[15] "Amazon ec2 service level agreement," https://aws.amazon.com/ec2/sla/ Accessed 06-Jun-2016.

[16] "Google compute engine service level agreement," https://cloud.google.com/compute/sla#definitions Accessed 10-Jun-2016.

[17] Z. Wu, M. Butkiewicz, D. Perkins, E. Katz-Bassett, and H. V. Madhyastha, "Spanstore: Cost-effective geo-replicated storage spanning multiple cloud services," in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 292–308.

[18] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman, and S. Shah, "Serving large-scale batch computed data with project voldemort," in *Proceedings of the 10th USENIX conference on File and Storage Technologies*. USENIX Association, 2012, pp. 18–18.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS Operating Systems Review*, vol. 41, no. 6. ACM, 2007, pp. 205–220.

[20] G. Linden, "Make Data Useful," https://sites.google.com/site/glinden/Home/StanfordDataMining.2006-11-28.ppt.

[21] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[22] A. Li, X. Yang, S. Kandula, and M. Zhang, "Cloudcmp: comparing public cloud providers," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 1–14.

[23] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart, "Next stop, the cloud: Understanding modern web service deployment in ec2 and azure," in *Proceedings of the 2013 conference on Internet measurement conference*. ACM, 2013, pp. 177–190.

[24] A. Bessani, M. Correia, B. Quaresma, F. André, and P. Sousa, "Depsky: dependable and secure storage in a cloud-of-clouds," *ACM Transactions on Storage (TOS)*, vol. 9, no. 4, p. 12, 2013.

[25] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 164–177, 2003.

[26] Microsoft, "Hyper-v," https://technet.microsoft.com/en-us/library/hh831531(v=ws.11).aspx Accessed 9-Jul-2016.

[27] "Kernel virtual machine (kvm)," http://www.linux-kvm.org/ Accessed 9-Jul-2016.

[28] "Linux containers," https://linuxcontainers.org/ Accessed 9-Jul-2016.

[29] V. Delgado, "Exploring the limits of cloud computing," 2010.

[30] R. Shea, F. Wang, H. Wang, and J. Liu, "A deep investigation into network performance in virtual machine based cloud environments," in *INFOCOM, 2014 Proceedings IEEE*. IEEE, 2014, pp. 1285–1293.

[31] G. Wang and T. E. Ng, "The impact of virtualization on network performance of amazon ec2 data center," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[32] J. Whiteaker, F. Schneider, and R. Teixeira, "Explaining packet delays under virtualization," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 38–44, 2011.

[33] K. Fraser, S. Hand, R. Neugebauer, I. Pratt, A. Warfield, and M. Williamson, "Safe hardware access with the xen virtual machine monitor," in *1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure (OASIS)*, 2004, pp. 1–1.

[34] H. Ballani, P. Costa, T. Karagiannis, and A. Rowstron, "Towards predictable datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 242–253.

[35] J. Schad, J. Dittrich, and J.-A. Quiané-Ruiz, "Runtime measurements in the cloud: observing, analyzing, and reducing variance," *Proceedings of the VLDB Endowment*, vol. 3, no. 1-2, pp. 460–471, 2010.

[36] P. S. I. Group, "Single root i/o virtualization," http://pcisig.com/ Accessed 6-Jul-2016.

[37] G. K. Lockwood, M. Tatineni, and R. Wagner, "Sr-iov: Performance benefits for virtualized interconnects," in *Proceedings of the 2014 Annual Conference on Extreme Science and Engineering Discovery Environment*. ACM, 2014, p. 47.

[38] M.-A. Kourtis, G. Xilouris, V. Riccobene, M. J. McGrath, G. Petralia, H. Koumaras, G. Gardikis, and F. Liberal, "Enhancing vnf performance by exploiting sr-iov and dpdk packet processing acceleration."

[39] M. Mitzenmacher, "The power of two choices in randomized load balancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 10, pp. 1094–1104, 2001.

[40] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[41] D. Paqué, "From symbolic execution to concolic testing," https://concurrency.cs.uni-kl.de/documents/Logics_Seminar_2014/SymbolicExecutionConcolicTesting.pdf Accessed 28-Jul-2016.

[42] ——, "From symbolic execution to concolic testing (slides)," https://concurrency.cs.uni-kl.de/documents/Logics_Seminar_2014/SymbolicExecutionConcolicTesting_slides.pdf Accessed 28-Jul-2016.

[43] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, "Exe: automatically generating inputs of death," *ACM Transactions on Information and System Security (TISSEC)*, vol. 12, no. 2, p. 10, 2008.

[44] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[45] C. Hale, "Java metrics library," 2015, https://github.com/dropwizard/metrics.

[46] G. Cormode, V. Shkapenyuk, D. Srivastava, and B. Xu, "Forward decay: A practical time decay model for streaming systems," in *Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on*. IEEE, 2009, pp. 138–149.

[47] Apache, "Cassandra, snitch types," 2016, https://docs.datastax.com/en/cassandra/2.0/cassandra/architecture/architectureSnitchesAbout_c.html.

[48] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *International Conference on Computer Aided Verification*. Springer, 2007, pp. 519–531.

[49] F. A. Manzano, "The symbolic maze!" 2010, https://feliam.wordpress.com/2010/10/07/the-symbolic-maze/.

[50] C. Cadar and D. Engler, "Execution generated test cases: How to make systems code crash itself," in *International SPIN Workshop on Model Checking of Software*. Springer, 2005, pp. 2–23.

[51] A. Markopoulou, F. Tobagi, and M. Karam, "Loss and delay measurements of internet backbones," *Computer Communications*, vol. 29, no. 10, pp. 1590–1604, 2006.

[52] S. Bucur, V. Ureche, C. Zamfir, and G. Candea, "Parallel symbolic execution for automated real-world software testing," in *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 183–198.

[53] R. S. Tsay, "Outliers, level shifts, and variance changes in time series," *Journal of forecasting*, vol. 7, no. 1, pp. 1–20, 1988.

[54] N. S. Balke, "Detecting level shifts in time series," *Journal of Business & Economic Statistics*, vol. 11, no. 1, pp. 81–92, 1993.

[55] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira, "Avoiding traceroute anomalies with paris traceroute," in *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*. ACM, 2006, pp. 153–158.

[56] D. L. Mills, "Internet time synchronization: the network time protocol," *Communications, IEEE Transactions on*, vol. 39, no. 10, pp. 1482–1493, 1991.

[57] V. Jacobson, "traceroute, 1989," *URL: ftp://ftp.ee.lbl.gov*.

[58] CISCO, "Per packet load balancing," http://www.cisco.com/c/en/us/td/docs/ios/12_0s/feature/guide/pplb.html Accessed 11-May-2016.

[59] Balakrishnan Chandrasekaran and Georgios Smaragdakis and Arthur Berger and Matthew Luckie and Keung-Chi Ng, "A Server-to-Server View of the Internet," in *Proceedings of ACM CoNEXT 2015*, Heidelberg, Germany, December 2015.

[60] "scales - Metrics for Python," https://github.com/Cue/scales.

[61] P. Francis, S. Jamin, C. Jin, Y. Jin, D. Raz, Y. Shavitt, and L. Zhang, "Idmaps: A global internet host distance estimation service," *IEEE/ACM Transactions On Networking*, vol. 9, no. 5, pp. 525–540, 2001.

[62] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iplane: An information plane for distributed services," in *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 367–380.

[63] "Scapy: packet manipulation library." http://www.secdev.org/projects/scapy/ Accessed 16-Sep-2016.

[64] J. Postel, "Internet control message protocol," 1981.

[65] M. C. Toren, "tcptraceroute," https://github.com/mct/tcptraceroute Accessed 16-Sep-2016.

[66] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with ycsb," in *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 2010, pp. 143–154.

[67] A. EC2, "Enabling enhanced networking with the intel 82599 vf interface on linux instances in a vpc," http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/sriov-networking.html Accessed 20-Aug-2016.

[68] XEN, "Credit scheduler," http://wiki.xen.org/wiki/Credit_Scheduler Accessed 31-Jul-2016.

[69] F. Zhou, M. Goel, P. Desnoyers, and R. Sundaram, "Scheduler vulnerabilities and attacks in cloud computing," *arXiv preprint arXiv:1103.0759*, 2011.

[70] L. Cherkasova, D. Gupta, and A. Vahdat, "Comparison of the three cpu schedulers in xen," *SIGMETRICS Performance Evaluation Review*, vol. 35, no. 2, pp. 42–51, 2007.

[71] T. Gleixner, "Cyclictest: A benchmark for real time linux kernel," https://rt.wiki.kernel.org/index.php/Cyclictest Accessed 9-Jul-2016.

[72] R. S. Boyer, B. Elspas, and K. N. Levitt, "Select - a formal system for testing and debugging programs by symbolic execution," *ACM SigPlan Notices*, vol. 10, no. 6, pp. 234–245, 1975.

[73] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on software engineering*, no. 3, pp. 215–222, 1976.

[74] S. Anand, C. S. Păsăreanu, and W. Visser, "Jpf–se: A symbolic execution extension to java pathfinder," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2007, pp. 134–138.

[75] K. Sen and G. Agha, "Cute and jcute: Concolic unit testing and explicit path model-checking tools," in *International Conference on Computer Aided Verification*. Springer, 2006, pp. 419–423.

[76] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, "Using model checking with symbolic execution to verify parallel numerical programs," in *Proceedings of the 2006 international symposium on Software testing and analysis*. ACM, 2006, pp. 157–168.

[77] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, vol. 23, no. 5, p. 279, 1997.

[78] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Păsăreanu, "Differential symbolic execution," in *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 2008, pp. 226–237.

[79] R. Sasnauskas, O. Landsiedel, M. H. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*. ACM, 2010, pp. 186–196.

[80] A. Vahdat, J. S. Chase, R. Braynard, D. Kostić, P. Reynolds, and A. Rodriguez, "Self-organizing subsets: From each according to his abilities, to each according to his needs," in *IPTPS*, ser. Lecture Notes in Computer Science, vol. 2429. Springer, 2002. ISBN 3-540-44179-4 pp. 76–84. [Online]. Available: http://dblp.uni-trier.de/db/conf/iptps/iptps2002.html#VahdatCBKRR02

[81] F. Dabek, R. Cox, F. Kaashoek, and R. Morris, "Vivaldi: A decentralized network coordinate system," in *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4. ACM, 2004, pp. 15–26.

[82] T. E. Ng and H. Zhang, "Predicting internet network distance with coordinates-based approaches," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 2002, pp. 170–179.

[83] J. Ledlie, P. Gardner, and M. I. Seltzer, "Network coordinates in the wild." in *NSDI*, vol. 7, 2007, pp. 299–311.

[84] B. Wong, A. Slivkins, and E. G. Sirer, "Meridian: A lightweight network location service without virtual coordinates," in *ACM SIGCOMM Computer Communication Review*, vol. 35, no. 4. ACM, 2005, pp. 85–96.

[85] M. J. Freedman, K. Lakshminarayanan, and D. Mazières, "Oasis: Anycast for any service." in *NSDI*, vol. 6, 2006, pp. 10–10.

[86] P. Wendell, J. W. Jiang, M. J. Freedman, and J. Rexford, "Donar: decentralized server selection for cloud services," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4, pp. 231–242, 2011.

[87] L. Suresh, M. Canini, S. Schmid, and A. Feldmann, "C3: Cutting tail latency in cloud data stores via adaptive replica selection," in *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, 2015.

[88] S. Agarwal, J. Dunagan, N. Jain, S. Saroiu, A. Wolman, and H. Bhogan, "Volley: Automated data placement for geo-distributed cloud services." in *NSDI*, 2010, pp. 17–32.

[89] J.-C. Bolot, "End-to-end packet delay and loss behavior in the internet," in *ACM SIGCOMM Computer Communication Review*, vol. 23, no. 4. ACM, 1993, pp. 289–298.

[90] V. Paxson, "End-to-end routing behavior in the internet," *Networking, IEEE/ACM Transactions on*, vol. 5, no. 5, pp. 601–615, 1997.

[91] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "Planetlab: an overlay testbed for broad-coverage services," *ACM SIGCOMM Computer Communication Review*, vol. 33, no. 3, pp. 3–12, 2003.

[92] J. S. Vitter, "Faster methods for random sampling," *Communications of the ACM*, vol. 27, no. 7, pp. 703–718, 1984.

[93] ——, "Random sampling with a reservoir," *ACM Transactions on Mathematical Software (TOMS)*, vol. 11, no. 1, pp. 37–57, 1985.

[94] C. C. Aggarwal, "On biased reservoir sampling in the presence of stream evolution," in *Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 607–618.

[95] K. Winstein and H. Balakrishnan, "Tcp ex machina: Computer-generated congestion control," in *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4. ACM, 2013, pp. 123–134.

[96] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson, "Tcp vegas: New techniques for congestion detection and avoidance," *SIGCOMM Comput. Commun. Rev.*, vol. 24, no. 4, pp. 24–35, Oct. 1994.

[97] J. C. Hoe, "Improving the start-up behavior of a congestion control scheme for tcp," in *ACM SIGCOMM Computer Communication Review*, vol. 26, no. 4. ACM, 1996, pp. 270–280.

[98] S. Ha, I. Rhee, and L. Xu, "Cubic: a new tcp-friendly high-speed tcp variant," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 64–74, 2008.

[99] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan, "An experimental study of the learnability of congestion control," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 479–490.

[100] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, "Data center tcp (dctcp)," in *ACM SIGCOMM computer communication review*, vol. 40, no. 4. ACM, 2010, pp. 63–74.

[101] C. Lee, C. Park, K. Jang, S. Moon, and D. Han, "Accurate latency-based congestion feedback for datacenters," in *2015 USENIX Annual Technical Conference (USENIX ATC 15)*, 2015, pp. 403–415.

[102] L. Cheng, C.-L. Wang, and F. C. Lau, "Pvtcp: Towards practical and effective congestion control in virtualized datacenters," in *2013 21st IEEE International Conference on Network Protocols (ICNP)*. IEEE, 2013, pp. 1–10.

[103] Y. Vigfusson, H. Abu-Libdeh, M. Balakrishnan, K. Birman, R. Burgess, G. Chockler, H. Li, and Y. Tock, "Dr. multicast: Rx for data center communication scalability," in *Proceedings of the 5th European conference on Computer systems*. ACM, 2010, pp. 349–362.

[104] Y.-H. Chu, S. G. Rao, S. Seshan, and H. Zhang, "A case for end system multicast," *Selected Areas in Communications, IEEE Journal on*, vol. 20, no. 8, pp. 1456–1471, 2002.

[105] A. Petlund, P. Beskow, J. Pedersen, E. S. Paaby, C. Griwodz, and P. Halvorsen, "Improving sctp retransmission delays for time-dependent thin streams," *Multimedia Tools and Applications*, vol. 45, no. 1-3, pp. 33–60, 2009.

[106] P. Arlos and M. Fiedler, "A method to estimate the timestamp accuracy of measurement hardware and software tools," in *Passive and Active Network Measurement*. Springer, 2007, pp. 197–206.

[107] J. Sommers and P. Barford, "An active measurement system for shared environments," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*. ACM, 2007, pp. 303–314.

[108] P. Orosz and T. Skopko, "Performance evaluation of a high precision software-based timestamping solution for network monitoring," *International Journal on Advances in Software*, vol. 4, no. 1, 2011.

[109] V. Paxson, "On calibrating measurements of packet transit times," in *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 1. ACM, 1998, pp. 11–21.

[110] S. B. Moon, P. Skelly, and D. Towsley, "Estimation and removal of clock skew from network delay measurements," in *INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 1999, pp. 227–234.

[111] L. Zhang, Z. Liu, and C. Honghui Xia, "Clock synchronization algorithms for network measurements," in *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, vol. 1. IEEE, 2002, pp. 160–169.

[112] L. De Vito, S. Rapuano, and L. Tomaciello, "One-way delay measurement: State of the art," *Instrumentation and Measurement, IEEE Transactions on*, vol. 57, no. 12, pp. 2742–2750, 2008.

[113] M. Shin, M. Park, D. Oh, B. Kim, and J. Lee, "Clock synchronization for one-way delay measurement: A survey," in *Advanced Communication and Networking*. Springer, 2011, pp. 1–10.

[114] M. Luckie, A. Dhamdhere, D. Clark, B. Huffaker *et al.*, "Challenges in inferring internet interdomain congestion," in *Proceedings of the 2014 Conference on Internet Measurement Conference*. ACM, 2014, pp. 15–22.

[115] "Linux kernel timestamping documentation," https://www.kernel.org/doc/Documentation/networking/timestamping.txt Accessed 6-Jul-2016.

[116] *recvmsg(2) Linux User's Manual*, July 2016.

[117] D. Miller, "ethtool," https://www.kernel.org/pub/software/network/ethtool/ Accessed 6-Jul-2016.

[118] "Tcpdump and libpcap," http://www.tcpdump.org/#documentation Accessed 6-Jul-2016.

Appendix A

Appendix

A.1 Timestamping in Linux

To understand the propagation behavior of packets traveling through a network, it is important to correlate the temporal and spacial domains. In other words, it is important to know where packets were at a given moment in time.

When a packet first arrives at the physical NIC, the NIC places the received frame into a circular buffer. The circular buffer is accessible to both the NIC and the OS's kernel. Next, the device's driver generates an interrupt to a CPU (assuming an interrupt based operation mode) or the CPU polls the NIC to see if there are packets available for it (in a polling mode of operation). At that moment, the kernel places the packet into its processing queue and puts a timestamp in the packet's buffer*. Now the packet is managed by the Linux networking stack and eventually the packet's contents are made available to a user-space application. The initial timestamp is preserved until the packet's contents can be accessed by the user-space application.

While kernel timestamping mitigates most of the delays associated with software packet processing, it still does not reflect the actual (true) time when the packet arrived at the NIC. To overcome this difference NIC based timestamping is required.

Finally, applications that are less sensitive to the precision of the timestamping or only use timestamping to identify the order in which an application processed the packets, often rely on user-space timestamping. The precision of user space timestamping is a function of the scheduling of the user space application.

The later in the processing pipeline that we delay the timestamping of the packet the less precision available. Moreover, the time intervals between each

*Note, regardless of the operational mode of the Linux driver (i.e., interrupt driven or polling), timestamping occurs at the same stage of the packet processing pipeline.

element of the pipeline depend upon the actual workload on the machine and the specific computer's hardware capabilities.

There are some parameters that affect the precision of timestamps: (i) the resolution of a clock used for the timestamping, and (ii) scheduling of when time is requested and measured (in the case of the competing processes). The resolution and stability of hardware clocks on the motherboard will also have an effect on the precision and maximal resolution of a time query [108].

The following sections outline techniques that can be used to perform timestamping using the Linux OS via the C/C++ kernel's socket API.

A.1.1 User Space Timestamping

The most basic approach of performing **user-space timestamping** is to execute `gettimeofday()` right after the `recv()` or `send()` calls. This allows an application to request OS-level time just after receiving or sending the packet. However, there are no guarantees that the two functions calls will be executed sequentially. With an increased load on a system, the delay between these two operations is expected to increase as the probability of the task being suspended and later being rescheduled increases. Note, that the user level timestamps only reflect the time when the packet becomes available to the user level application or when the packet has been sent to the OS's networking stack. Such timestamps do not reflect the time when the packets arrived to or leave the physical NIC.

A.1.2 Kernel Space Timestamping

Linux uses the socket API to configure and provide access to kernel level and hardware timestamps. A comprehensive overview can be found in the Linux kernel documentation of the control interface [115], furthermore, `timestamping.c` located in the Linux source code can provide a good example of the usage patterns.

Kernel level timestamping can be enabled by passing the `SO_TIMESTAMP`, `SO_TIMESTAMPNS`, or `SO_TIMESTAMPING` flags to the `setsockopt()` system call. In this thesis, we relied on the `SO_TIMESTAMPING`, which is the most feature reach and flexible option. This option enables packet timestamping both on reception and transmission and provides access to the hardware timestamps[†].

A.1.2.1 Accessing Kernel Space Timestamps Using Datagrams

Figure A.1 demonstrates an example of working with kernel level timestamp using UDP. The `SO_TIMESTAMPING` is passed to the socket options in line 12. The rest of the timestamping specific options are passed via special flags in line 13.

[†]This option requires hardware support at the NIC, see Section A.1.3.

```
1  socklen_t sockfd;
2  sockfd = socket(AF_INET, SOCK_DGRAM, 0);
3
4  // socket initialization goes here
5
6  int SOCKOPT_FLAGS = SOF_TIMESTAMPING_TX_SOFTWARE |
7                      SOF_TIMESTAMPING_RX_SOFTWARE |
8                      SOF_TIMESTAMPING_SOFTWARE;
9
10 int err = setsockopt(sockfd,
11                       SOL_SOCKET,
12                       SO_TIMESTAMPING,
13                       &SOCKOPT_FLAGS,
14                       sizeof(int));
15 if (err == -1) {
16     close(sockfd);
17     printf("Error msg:[%s]\n", strerror(errno));
18     return -1;
19 }
```

Figure A.1: Socket options for enabling kernel level timestamping.

The `SOF_TIMESTAMPING_RX_SOFTWARE` and `SOF_TIMESTAMPING_TX_SOFTWARE` enable software timestamping when packets enter and leave the kernel through the NIC respectively. Finally, the `SOF_TIMESTAMPING_SOFTWARE` enables the reporting of the software timestamps by the kernel.

The access to the generated timestamps is provided via the ancillary data structure of `recvmsg` [116], pointed to by the `msg_control` pointer (see Figure A.2). This is done by passing `msghdr` structure into `recvmsg`. The `msghdr` data structure contains both a buffer to receive a packet (line 15) and a buffer to receive the control data structure (line 13). The control data structure is used as a feedback mechanism to receive ancillary messages from the kernel. One of the control messages contains the packet's timestamp.

The `recvmsg` call is also used to get packet's transmission timestamps. If the `SOF_TIMESTAMPING_SOFTWARE` option is set, then outgoing packets are being cloned and looped back to the socket's error queue with the sent timestamp annotated. Line 30 in Figure A.2 demonstrates how `msghdr` data structure can be filled up after a packet has been sent.

The `msghdr` contains a set of control messages. Figure A.3 demonstrates how control messages can be iterated and how the timestamp is being retrieved. The timestamp is retrieved via the `scm_timestamping` data structure which is an array of three `timespec` structures (see line 14). The software timestamp is located in the first cell of that array.

```c
int rcv_msg(int sockfd, void *buf, size_t len, int flags) {
    struct msghdr msg;
    memset(&msg, 0, sizeof(msg));

    struct iovec entry;
    struct sockaddr_in from_addr;
    struct {
        struct cmsghdr cm;
        // buffer for ancillary messages
        char control[128];
    } control;

    msg.msg_control = &control;
    msg.msg_controllen = sizeof(control);
    entry.iov_base = buf;
    entry.iov_len = sizeof(len);
    msg.msg_iov = &entry;
    msg.msg_iovlen = 1;

    int rcv_bytes = recvmsg(sockfd, &msg, 0);
    // read timestamp from msg data structure here
}
```

```c
int send_msg(int sockfd, const void *buf, size_t len,
             int flags, const struct sockaddr *dest,
             socklen_t addrlen) {
    // send the data
    int sent_bytes = sendto(sockfd, buf, len, flags,
                            dest, addrlen);
    // check error queue for timestamp
    int rcv_bytes = recvmsg(sockfd, &msg, MSG_ERRQUEUE);
    // read timestamp from msg data structure here
}
```

Figure A.2: Accessing kernel timestamps via `sendmsg` when receiving and sending messages. Error checking is omitted.

```
uint64_t get_tstamp_from_control_msg(struct msghdr & msg) {
    struct cmsghdr * cmsg;
    uint64_t k_tstamp = 0;

    for (cmsg = CMSG_FIRSTHDR(&msg);
         cmsg;
         cmsg = CMSG_NXTHDR(&msg, cmsg) ) {

        if ( (cmsg->cmsg_level == SOL_SOCKET) &&
             (cmsg->cmsg_type == SO_TIMESTAMPING) &&
             (cmsg->cmsg_len == EXPECTED_K_TSTMP_SIZE)) {

            const struct timespec * stamp =
                (struct timespec *)CMSG_DATA(cmsg);

            k_tstamp = timespec_to_uint64(stamp);
            return k_tstamp;
        }
    }
    return k_tstamp;
}
```

Figure A.3: Extracting kernel timestamps from the msghdr data structure.

```
uint64_t get_kernel_recv_tstamp(int sockfd) {
    struct timeval tv_ioctl;
    tv_ioctl.tv_sec = 0;
    tv_ioctl.tv_usec = 0;

    int err = ioctl(sockfd, SIOCGSTAMP, &tv_ioctl);
    if (err == -1) {
        printf ( "Error ioctl:[%s]\n", strerror(errno));
        return 0;
    }
    return timeval_to_uint64(&tv_ioctl);
}
```

Figure A.4: Accessing kernel timestamps via ioctl system call.

Alternatively, timestamps can also be accessed via a call to `ioctl` (see Figure A.4). By specifying `SIOCGSTAMP` flag, the `ioctl` system call retrieves the timestamp of the last packet that has been sent or received through the specified socket. Note, that the `SIOCGSTAMP` flag in `ioctl` call is mutually exclusive with the `SO_TIMESTAMPING` socket option. Only one of these options can be used at a time. However, using `recvmsg` approach appears to be more efficient as it requires only a single system call when receiving messages.

A.1.2.2 Accessing kernel level timestamps in byte streams

Special care has to be taken when timestamping stream based protocols (e.g., TCP). Such protocols do not maintain the notion of packet boundaries, but instead, send data as a stream of bytes - which is subsequently sent as segments. In TCP, a "logical packet" as perceived by application, can be merged, split or rearranged at the IP layer, therefore matching timestamps with the data is difficult. The generic approach implemented in Linux kernel operates on a per buffer level and timestamps a buffer when the last byte has been passed from the kernel to the NIC driver via the network driver (in the case of outgoing packets). Many corner cases are possible and might require additional logic to implement. Currently, Linux supports TCP timestamping only for the TX queue [115][‡]. If the precision is not critical, a user-space timestamping can be used to alleviate this complexity, after parsing the byte stream into logical packets (for example, as currently implemented in Cassandra).

A.1.3 Hardware Timestamping

Hardware timestamping is dependent on the NIC's capabilities and requires root permissions to be initiated.

The first step is to configure the NIC's driver to enable hardware timestamping. This is done by calling `ioctl(SIOCSHWTSTAMP)` with a pointer to a `ifreq` data structure which points to the `hwtstamp_config` via `ifr_data` (lines 1, 5, and 11 in Figure A.5).

Line 7 enables hardware timestamping for outgoing packets. Line 8 sets the RX queue filter for timestamping incoming packets. The RX queue filter describes the type of packets to timestamp. Note, NIC's driver configuration for hardware timestamps should be seeing as a "requested" configuration. The driver might modify the timestamp options according to NIC's capabilities, thus the returned

[‡]TCP timestamping is relatively new feature in the Linux's kernel introduced in version 3.19. This also suggests that the code and features might change, thus the documentation and the source code of the particular kernel needs to be examined before attempting TCP timestamping.

```
struct ifreq hwtstamp;
struct hwtstamp_config hwconfig, hwconfig_requested;

strncpy(hwtstamp.ifr_name, "eth0", sizeof(hwtstamp.ifr_name));
hwtstamp.ifr_data = (void *)&hwconfig;

hwconfig.tx_type = HWTSTAMP_TX_ON;
hwconfig.rx_filter = HWTSTAMP_FILTER_PTP_V1_L4_SYNC;
hwconfig_requested = hwconfig;

if (ioctl(sockfd, SIOCSHWTSTAMP, &hwtstamp) < 0) {
  if ((errno == EINVAL || errno == ENOTSUP) &&
    hwconfig_requested.tx_type == HWTSTAMP_TX_OFF &&
    hwconfig_requested.rx_filter == HWTSTAMP_FILTER_NONE)
    printf("Hardware time stamping not possible\n");
  else
    bail("SIOCSHWTSTAMP");
}
printf("tx_type %d requested, got %d;
        rx_filter %d requested, got %d\n",
       hwconfig_requested.tx_type, hwconfig.tx_type,
       hwconfig_requested.rx_filter, hwconfig.rx_filter);

// Set socket options
SOCKOPT_FLAGS = SOF_TIMESTAMPING_TX_HARDWARE |
                SOF_TIMESTAMPING_RX_HARDWARE |
                SOF_TIMESTAMPING_RAW_HARDWARE;

int err = setsockopt(sockfd,
                     SOL_SOCKET,
                     SO_TIMESTAMPING,
                     &SOCKOPT_FLAGS,
                     sizeof(int));
```

Figure A.5: NIC's driver configuration and hardware socket options to enable hardware timestamping. Based on the code snipped from `timestamping.c` from Linux's source code documentation (version 3.16).

`hwtstamp_config` data structure from the `ioctl` call should be validated against the initially requested configuration (line 19).

Similarly to the kernel level timestamping, hardware timestamping requires `SO_TIMESTAMPING` option along with specific flags (line 25). Timestamps are also read via the `recvmsg` structure, however, in contrast to kernel level timestamps, the hardware timestamps located at the third cell of the `scm_timestamping` data structure (see line 14 in Figure A.3).

Ethtool[117] with option `-T` can be used to verify NIC's capabilities to perform hardware timestamping. Tcpdump[118] can perform hardware timestamping while capturing both the incoming and outgoing traffic (using option `-j`). Tcpdump is implemented on top of the pcap library, which in turn uses the standard Linux API to configure packet timestamping as discussed previously.

Unfortunately, today it appears to be impossible activate hardware assisted timestamping in a third party cloud provider (particularly EC2) for the following reasons. First, hardware timestamping requires root permissions on the host machine (dom0) to initialize NIC driver for timestamping. Second, the current version (2.14.2) of the *ixgbevf* used in EC2 does not provide an API to turn on hardware timestamping. While we investigated the case of EC2, it is likely that the same reasons will be relevant for other cloud providers.

A.2 Intercepting Linux Socket API calls

`LD_PRELOAD` allows to specify a shared library that will be preloaded *before* any other library when running an executable. In practice this allows to replace the content of any linked libraries. For example, Figure A.6 demonstrates an example of replacing standard `sendmsg()` call with a modified implementation. This technique works equally well with Java, as it uses the same native Linux call to execute operations on sockets. When an application calls the *sendmsg* function, the modified version of the function will be executed. On line 20 it is possible to intercept the send call and introduce our own custom logic.

Lines 26 and 27 show how to compile the shared library and how to preload it before any other libraries. Socket API is part of the Linux kernel and it is compiled using C. Therefore, a C complier (e.g., `gcc`) or `extern "C"` should be used to avoid function name mangling, thus the symbols in the shared library will match the API call performed by an application.

A set of useful commands to debug and understand this process includes: (i) `export LD_DEBUG=all` shows the process of loading and searching execution symbols for all executables, (ii) `objdump -T socket.so` or `nm -D socket.so` will show the symbols compiled in the shared library.

```
1
2  #define _GNU_SOURCE
3  #include <stdio.h>
4  #include <assert.h>
5  #include <sys/socket.h>
6  #include <sys/types.h>
7  #include <dlfcn.h>
8
9  typedef ssize_t (*o_sendmsg_ptr)
10     (int sockfd, const struct msghdr *msg, int flags);
11
12 // Pointer to the original sendmsg funciton
13 o_sendmsg_ptr o_sendmsg =
14     (o_sendmsg_ptr)dlsym(RTLD_NEXT, "sendmsg");
15
16 ssize_t
17 sendmsg(int sockfd, const struct msghdr *msg, int flags) {
18     assert (o_sendmsg != NULL);
19
20     // Instrumentation code goes here...
21
22     return o_sendmsg(sockfd, msg, flags);
23 }
24
25 # To compile and run:
26 $ gcc -Wall -fPIC -shared socket.c -o socket.so -ldl
27 $ LD_PRELOAD=./socket.so ./cassandra -f
```

Figure A.6: Using `LD_PRELOAD` to modify functionality of the Linux socket API.

A.3 EC2 Schedule Probe

Figure A.7 shows core elements of the interval measurement probe used to infer Amazon's EC2 scheduling configurations. Intervals below 1 ms preserved in a histogram. Timestamps and the lengths of the intervals above 1 ms are stored in a buffer and later written to disk (not shown in the code). The relative time distance between long intervals allows to estimate the duration of the interruption and duration of the processing.

The bottom part of Figure A.7 shows the script that was launched periodically using `cront` jobs. Lines 39 and 42 ping the workload and the IMP to one CPU core. On Line 44 we use `chrt` system call to change the default Linux scheduler for the IMP's PID to SCHED_FIFO (argument `-f`); the priority of the process is set to highest (value 99).

```c
#define BILLION 1000000000L
#define HISTOGRAM_SIZE 1000000
#define MAX_INT_SAVE 5000000
#define TDELTA(start, end) BILLION * (end.tv_sec - start.tv_sec) \
        + end.tv_nsec - start.tv_nsec
#define T2NANO(start) BILLION * start.tv_sec + start.tv_nsec
uint64_t MAX_COLL_TIME_NS;
uint64_t * long_ints;
uint64_t hist[HISTOGRAM_SIZE] = {0};
int main(int argc, char**argv) {
    int sleep_usec = 10; int nmins = 19; uint64_t t_delta_ns;
    MAX_COLL_TIME_NS = 60*BILLION*nmins;
    long_ints = (uint64_t*)malloc(sizeof(uint64_t)*MAX_INT_SAVE);
    struct timespec t_experiment_start, t_last, t_now;
    clock_gettime(CLOCK_MONOTONIC, &t_experiment_start);
    clock_gettime(CLOCK_MONOTONIC, &t_last);
    int n_intervals_saved = 0;
    for (; n_intervals_saved < MAX_INT_SAVE - 1 ; ) {
        usleep(sleep_usec);
        clock_gettime(CLOCK_MONOTONIC, &t_now);
        t_delta_ns = TDELTA(t_last, t_now);
        if (t_delta_ns < HISTOGRAM_SIZE) {
            hist[t_delta_ns] += 1;
            t_last = t_now;
        } else {
            *long_ints = T2NANO(t_last);
            long_ints++; n_intervals_saved++;
            *long_ints = t_delta_ns;
            long_ints++; n_intervals_saved++;
            t_last = t_now;
        }
        if (TDELTA(t_experiment_start, t_now) > MAX_COLL_TIME_NS)
            break;
    }
```

```bash
#!/bin/bash
cd "$(dirname "$0")"
for i in `seq 1 $1`
do
    taskset -c $(($i-1)) ./easyloop &
    LAST_PID=$!
    nice -n 19 -p $LAST_PID
    taskset -c $(($i-1)) ./time_sched &
    LAST_PID=$!
    sudo chrt -f -p 99 $LAST_PID
    chrt -p $LAST_PID
done
```

Figure A.7: Interval measurement probe (on the top) and the bash script that sets priorities and configures Linux scheduler (on the bottom).

A.4 UDP RTT Probe

This section, illustrates the source code of the UDP probe that was used to measure WAN latency across EC2 datacenters.

```
#define BORDER_VALUE 1122334455

uint64_t getTimeNowUSec() {
    static struct timeval time;
    gettimeofday(&time, 0);
    return (uint64_t)
        (time.tv_sec*(1000.0*1000.0) + time.tv_usec);
}
struct PingMessage {
    uint32_t border_value;
    uint32_t id;
    uint64_t time_usec;
};
socklen_t          sockfd_send;
socklen_t          sockfd_rcv;
socklen_t          len;
struct sockaddr_in to_addr;
struct sockaddr_in from_addr;
const int MSG_SIZE = (sizeof(struct PingMessage));

int getDataFromSocket(struct PingMessage * msg_in,
                      socklen_t socketfd,
                      struct sockaddr * saddr,
                      socklen_t &len) {
    ssize_t total_bytes_rcv = 0;
    total_bytes_rcv = recvfrom(
        socketfd, msg_in, MSG_SIZE, 0, saddr, &len);
    if (total_bytes_rcv == -1) {
       printf("recv Error, ret=%i, %s\n",
          (int)total_bytes_rcv, strerror(errno));
       return -1;
    } else if (total_bytes_rcv < sizeof(msg_in)) {
       printf("rcv Warning Partial message \n");
       return 1;
    } else if (msg_in->border_value != BORDER_VALUE) {
       printf("rcv Warning Garbage content \n");
       return 1;
    }
    return 0;
}
void *tServerReceiver(void *ptr) {
    struct PingMessage msg_in;
    double delta_time = 0;
    while (true) {
```

```
45        int err = 0;
46        err = getDataFromSocket(&msg_in, sockfd_rcv,
47                  (struct sockaddr *) &from_addr, len);
48        if (err == -1) {
49           printf("Exiting::tServerReceiver \n");
50           break;
51        } else if (err > 0) {
52           continue;
53        }
54        delta_time = getTimeNowUSec()-msg_in.time_usec;
55        printf("Srvr::rcvd  %i %ld.%ld %f\n", msg_in.id,
56               msg_in.time_usec /(1000*1000),
57               msg_in.time_usec % (1000*1000),
58               (double)(delta_time / 1000.0));
59     }
60 }
61 void startServerPingSender(int ping_frequency) {
62     assert(ping_frequency >= 0);
63     struct PingMessage msg_out;
64     msg_out.border_value = BORDER_VALUE;
65     uint32_t msg_index = 0;
66     double delta_time;
67     double ping_interval_us = (1000*1000) / ping_frequency;
68     while (true) {
69         msg_out.id = msg_index++;
70         msg_out.time_usec = getTimeNowUSec();
71         int err = 0;
72         err = (int)sendto(sockfd_send, &msg_out,
73                           sizeof(msg_out), 0,
74                           (struct sockaddr *) &to_addr,
75                           sizeof(to_addr));
76         if (err == -1) {
77             printf("snd Error, ret=%i, %s\n",
78                    (int) err, strerror(errno));
79             break;
80         }
81         delta_time = getTimeNowUSec() - msg_out.time_usec;
82         if (delta_time < ping_interval_us) {
83             usleep(ping_interval_us - delta_time);
84         }
85     }
86 }
87 void startRepeater(int log_at_rep) {
88     struct PingMessage msg;
89     while(true) {
90         int err = 0;
91         err = getDataFromSocket(&msg, sockfd_rcv,
92                   (struct sockaddr *)&from_addr, len);
93         if (err == -1) {
```

```
94                printf("Exiting::tClientReceiver \n");
95                break;
96            } else if (err > 0) {
97                continue;
98            }
99            sendto(sockfd_send, &msg, MSG_SIZE, 0,
100                   (struct sockaddr *)&to_addr, sizeof(to_addr));
101           if (log_at_rep) {
102               printf("Rptr::frwd  %i -1 -1\n", msg.id);
103           }
104       }
105   }
106   int main(int argc, char**argv) {
107       // ... Arguments parsing omitted ....
108
109       char type = 0;
110       char * from_ip, * to_ip;
111       int from_port = 0, to_port = 0;
112       int ping_frequency = 0;
113       int log_at_rep = 0;
114       char c;
115       int long_index = 0;
116       sockfd_send = socket(AF_INET, SOCK_DGRAM, 0);
117       bzero(&to_addr, sizeof(to_addr));
118       to_addr.sin_family = AF_INET;
119       to_addr.sin_addr.s_addr = inet_addr(to_ip);
120       to_addr.sin_port = htons(to_port);
121
122       sockfd_rcv = socket(AF_INET, SOCK_DGRAM, 0);
123       bzero(&from_addr, sizeof(from_addr));
124       from_addr.sin_family = AF_INET;
125       from_addr.sin_addr.s_addr = htonl(INADDR_ANY);
126       from_addr.sin_port = htons(from_port);
127       bind(sockfd_rcv, (struct sockaddr *)&from_addr, sizeof(from_addr));
128
129       if (type == 's') {  // Server Type
130           pthread_t t_receiver;
131           int iret1;
132           iret1 = pthread_create(&t_receiver, NULL,
133                                  tServerReceiver, (void*) sockfd_rcv);
134           if (iret1) {
135               exit(EXIT_FAILURE);
136           }
137           startServerPingSender(ping_frequency);
138           pthread_join(t_receiver, NULL);
139       } else if (type == 'r') {  // Repeater Type
140           startRepeater(log_at_rep);
141       }
142   }
```