

ANALYZING DATA-CENTER APPLICATION PERFORMANCE VIA
CONSTRAINT-BASED MODELS

Junhua Yan

A thesis submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Computer Science.

Chapel Hill
2023

Approved by:

Jasleen Kaur

Van Jacobson

Colin Raffel

F. Donelson (Don) Smith

Kevin Jeffay

© 2023
Junhua Yan
ALL RIGHTS RESERVED

ABSTRACT

Junhua Yan: Analyzing Data-center Application Performance Via Constraint-based Models
(Under the direction of Jasleen Kaur)

Hyperscale Data Centers (HDCs) are the largest distributed computing machines ever constructed. They serve as the backbone for many popular applications, such as YouTube, Netflix, Meta, and Airbnb, which involve millions of users and generate billions in revenue. As the networking infrastructure plays a pivotal role in determining the performance of HDC applications, understanding and optimizing their networking performance is critical.

This thesis proposes and evaluates a constraint-based approach to characterize the networking performance of HDC applications. Through extensive evaluations conducted in both controlled settings and real-world case studies within a production HDC, I demonstrated the effectiveness of the constraint-based approach in handling the immense volume of performance data in HDCs, achieving tremendous dimension reduction, and providing very useful interpretability.

To my parents, Changhui Qi and Chao Sun,
whose unwavering love, support, and sacrifices have been the bedrock of my journey.

ACKNOWLEDGEMENTS

Firstly, I would like to express my heartfelt gratitude to my parents, Chao Sun and Changhui. They have consistently provided unwavering support throughout my doctoral journey.

I extend my sincere thanks to my advisors, Dr. Jasleen Kaur from UNC and Van Jacobson from Google. Under your guidance, I learned invaluable lessons on problem-solving approaches, solution formulation, idea execution, and effective communication. It is an immense honor to have been your student. I am also grateful to my committee members for their time and dedication in discussing my research ideas, reviewing my drafts, and providing constructive feedback.

I would like to acknowledge my labmate at UNC, Hasan Faik Alan. His commitment to research excellence has greatly influenced me, especially during our early interactions when I began my PhD journey. Hasan has been immensely helpful in both coursework and research, starting from the very first socket programming assignment in Prof. Kaur's class.

A special thank you goes to the department staff, including Denise, Missy, Janet, Bil, and numerous others. Your efforts have created an incredible environment within the department, fostering support and guidance for all of us.

I am grateful to my mentors and teammates at Google: Van Jacobson, Yuchung Cheng, Yousuk Seung, Mubashir Adnan Qureshi, Soheil Hassas Yeganeh, Neal Cardwell, David Wetherall, Nandita Dukkupati. The design and implementation of Fathom have been instrumental in enabling my work. I appreciate your willingness to address my inquiries, provide insightful guidance for implementing the pipeline, and offer valuable feedback on my case studies. I consider myself extremely fortunate to have become a part of the team since my initial internship in the summer of 2019. Although interning for more than two years may not have been anticipated, I consider it a stroke of luck. I extend my thanks to David and Nandita for their assistance in reviewing and approving my dissertation, enabling me to share my work with you all today.

To my friends, I am grateful for your presence in my life. I would like to thank my roommates, Yi Li and Zhen Wei, for inspiring me to become a better cook. After all, there's hardly anything that can't be solved with the magic of a delicious home-cooked meal. I express my gratitude to Meng Zou, Weixing Zhou, and Qianwen Yin for joining me in new experiences, sharing your thoughts, introducing me to captivating movies and books, and keeping me motivated and inspired. Lastly, to all my friends, thank you for providing companionship throughout these years.

The incredible support I received from those around me was an incredible source of motivation throughout my doctoral journey. I am truly grateful for their presence and encouragement. Thank you.

TABLE OF CONTENTS

LIST OF TABLES	xiii
LIST OF FIGURES	xvi
CHAPTER 1: INTRODUCTION	1
1.1 Understanding the Network Performance of HDC Applications is Critical	1
1.1.1 The Significance of Understanding the Network Performance of HDC Applications	1
1.1.1.1 Task 1: Assessment	2
1.1.1.2 Task 2: Planning	3
1.1.1.3 Task 3: Diagnosis	3
1.1.2 Common Characteristics of HDC Applications	4
1.1.2.1 Distributed	4
1.1.2.2 Diverse Application Workload & Requirements	5
1.1.2.3 Diverse Infrastructures	6
1.1.2.4 Coupled	6
1.1.3 Challenges in Understanding the Network Performance of HDC Applications	7
1.2 Performance Monitoring in HDCs: State-Of-The-Art	9
1.3 Our Approach: Analyzing HDC Application Network Performance Via Constraints-based Models	10
1.3.1 Our Goal	10
1.3.2 Implementation	11
1.3.3 GMM Evaluation	11
1.3.4 Enhancements	13

1.4	Thesis Organization	14
CHAPTER 2: RELATED WORK		15
2.1	Performance Monitoring/Tracing	15
2.2	Anomaly Detection and Characterization	19
2.2.1	Time-series Based Approaches	21
2.2.2	Performance Statistics/Traffic Features Based Approaches	23
2.2.2.1	Heuristic Rules	24
2.2.2.2	Statistical Tools	25
2.2.2.3	Learning-based Techniques	26
2.2.2.4	Rollout Impact Analysis	28
2.2.3	Dependency-graph Modeling	29
2.2.4	Limitations	32
CHAPTER 3: OUR APPROACH: CONSTRAINT-BASED MODELS BASED ON REMOTE PROCEDURE CALL TELEMETRY		34
3.1	The Goal: Constraint-Based Modeling	34
3.2	Performance Data: Remote Procedure Call Telemetry	37
3.2.1	How Constraints Manifest in HDCs	37
3.2.2	A Vantage Point for Observing Constraints: Remote Procedure Call	39
3.2.2.1	The RPC Workflow	40
3.2.2.2	Potential Constraints for an RPC	41
3.3	Interpretable Modeling: Gaussian Mixture Models (GMMs)	44
3.3.1	Background: Gaussian Mixture Models	44
3.3.2	Interpretable Modeling	44
CHAPTER 4: THE MODELING PIPELINE		48
4.1	GMM Modeling	49
4.1.1	Feature Selection	49

4.1.2	Model Selection	51
4.1.2.1	Goodness of Fit	53
4.1.2.2	Goodness of Separation	54
4.1.2.3	Goodness of Interpretation	55
4.1.2.4	Model Selection Approach Summary	61
4.1.3	Background Removal	63
4.1.3.1	Our Approach	66
4.1.3.2	Validation with a Public Dataset	68
4.2	Analysis of GMM blobs	72
4.2.1	Performance Characterization	73
4.2.2	Categorical Composition Analysis	74
4.2.3	Blob Matching	76
4.3	Conclusion	77
CHAPTER 5: EVALUATIONS ON THE CLOUDLAB EMULATION PLATFORM.....		78
5.1	Experimental Methodology	78
5.1.1	Traffic Characteristics	78
5.1.2	Testbed Topology	79
5.2	Performance Tracing on CloudLab	80
5.2.1	Monitoring Tools	80
5.2.1.1	SNMP (Switches)	80
5.2.1.2	Ifconfig (Clients & Servers)	81
5.2.1.3	Tcpdump (Clients & Servers)	81
5.2.2	Performance Metrics	81
5.2.2.1	Delay Metrics	82
5.2.2.2	Rate Metrics	83
5.2.2.3	Volume Metrics	83

5.2.2.4	Cross-traffic Information	85
5.2.3	Performance Constraints on CloudLab	85
5.3	Use Case 1: Understanding the Performance of Cache Application	86
5.3.1	What Can Be Learned from Traditional Approaches	87
5.3.2	What Can Be Learned Using GMM Analysis	92
5.3.3	Validating Constraints with Ground Truth Information	93
5.3.4	GMMs vs. kMeans vs. DBSCAN	98
5.4	Use Case 2: Analyzing the Impact of Configuration Factors on RPC Performance ..	100
5.4.1	Experiment 1: The Impact of Generic Segmentation Offloading (GSO)	100
5.4.1.1	Background: GSO	101
5.4.1.2	What Can Be Learned from Traditional Analysis	101
5.4.1.3	What Can Be Learned Using GMM Analysis	103
5.4.1.4	Validating Constraints Using Ground Truth Information	106
5.4.2	Experiment 2: The Impact of HTTP Persistence	107
5.4.2.1	The Impact of HTTP Keep-Alive with Multiple Connections	109
5.4.3	Experiment 3: The Impact of Communication Patterns	117
5.5	Use Case 3: Detecting Performance Anomalies	118
5.6	Conclusion	121
CHAPTER 6: CASE STUDIES FROM A PRODUCTION HDC		123
6.1	RPC Performance Instrumentation: Fathom	123
6.1.1	Performance Metrics Collected	124
6.1.2	Categorical Attributes Collected	126
6.1.3	RPC Sampling	127
6.1.4	t -Digests: Aggregation that Preserves Distributions	129
6.1.5	Training Features from Fathom	130
6.2	Case Study 1: Simpson's Paradox in RPC Telemetry	131

6.2.1	How GMMs Help	135
6.3	Case Study 2: Evaluating the Performance of a Major Service	137
6.3.1	GMM Results	138
6.3.1.1	GMM Distinguishes Performance Constraints	141
6.3.1.2	Categorical Attributes Influence the Performance Behavior of a GMM Blob	143
6.4	Case Study 3: Understanding Impact of Infrastructure Change: Congestion Control	145
6.4.1	Background: TCP Congestion Control	146
6.4.2	The Picture Without GMM Analysis	146
6.4.3	How Does GMM Help Understand Performance <i>Before</i> the Infrastructure Change?	147
6.4.4	How Does <i>newCC</i> Impact Networking Constraints Experienced by <i>J</i> in <i>StoreService</i> ?	149
6.4.5	Identifying a Needle in the Haystack	151
6.5	Case Study 4: Infrastructure Upgrade Planning	152
6.5.1	How Can GMM Inform Planning of Future Upgrades?	154
6.5.2	Discussion	154
6.6	Case Study 5: Root Cause Analysis of Performance Degradation	155
6.6.1	How Can GMM Analysis Help Separate Traffic With Degraded Networking Performance?	156
6.6.2	How Can GMM Analysis Shed Light on the Root Cause?	157
6.6.3	Discussion	158
6.7	Conclusion	159
CHAPTER 7: DATA COLLECTION INSTRUMENTATION MODIFICATIONS AND FOLLOW-UP CASE STUDIES		160
7.1	Issue 1: The Impact of RPC Size on Performance	160
7.1.1	The Network Performance of Small vs. Large RPCs	160
7.1.1.1	Background: Bandwidth and Delay	160

7.1.1.2	Background: Physical Constraints Limiting the Performance of Network Transfers	162
7.1.1.3	What Is Relevant for Small vs. Large RPCs.....	162
7.1.2	Potential Issues with Disproportional Ratios of Small and Large RPCs	163
7.1.3	Modifications to Fathom	164
7.2	Issue 2: Per-RPC vs. Per-Packet Delivery Rate	166
7.3	Case Study 1: Assess the Impact of Protective Load Balancing (PLB) Deployment ..	167
7.4	Case Study 2: Evaluate the Fleetwide Performance of Machine Learning Service in HDCs.....	170
7.4.1	Issue 1: Job Name Analysis	171
7.4.2	Issue 2: Match GMM blobs in Different DataCenter Clusters.....	172
7.4.3	What Can Be Learned from the Fleet-Wide Analysis.....	175
7.5	Conclusion.....	180
CHAPTER 8: CONCLUSIONS AND FUTURE WORK		181
8.1	Conclusions	181
8.2	Role of Constraint-based Modeling in Achieving Different Network Health Monitoring Goals	181
8.3	Limitations and Future Work	184
APPENDIX A: MONITORING TOOLS ON CLOUDLAB		187
A.1	SNMP	187
A.2	Ifconfig	187
A.3	Tcpdump	188
APPENDIX B: ADDITIONAL CASE STUDIES ON CLOUDLAB TESTBED.....		189
B.1	Experiment 1: The Impact of HTTP Keep-Alive with a Single Connection	189
B.2	Experiment 2: The Impact of Communication Patterns	193
REFERENCES		199

LIST OF TABLES

Table 3.1 – Constraint types and their associated performance metrics in Figures 3.1 and 3.2.	43
Table 4.1 – An example of two GMMs and their categorical attributes in each blob (b_n).....	57
Table 4.2 – Percentages of RPCs for different destination users in blobs C , C_1 , and C_2	61
Table 4.3 – Percentages of RPC measurements for different destination users in blobs (A , A_1 , A_2 , B , B_1 and B_2) in Figure 4.9.....	64
Table 4.4 – The Mahalanobis distance within/between blobs in Figure 4.12(a).	69
Table 4.5 – The Mahalanobis distance within/between blobs Figure 4.14(a).	71
Table 4.6 – The Mahalanobis distance between blobs in Figure 4.15(a).	71
Table 5.1 – A list of delay-related performance metrics extracted from pcap files.	83
Table 5.2 – Types of constraints each RPC may experience in the testbed.....	86
Table 5.3 – The most distinguishing metric between GMM blobs based on the Wasserstein distance.	92
Table 5.4 – Types of constraints experienced by RPCs based on GMMs.	93
Table 5.5 – Average performance of RPCs in each blob.....	104
Table 5.6 – Comparison of average performance with and without HTTP Keep-Alive for multiple connections.....	110
Table 5.7 – Distance measure of the 2-dimensional Wasserstein distance between blobs in two GMMs.....	110
Table 5.8 – Top 5 performance metrics that distinguish the blobs in each GMMs with multiple connections.	111
Table 5.9 – Top 3 performance metrics that distinguish each blob in the GMM when HTTP Keep-Alive is disabled.	111
Table 5.10 –Cross traffic and RPC size information for each blob when HTTP Keep-Alive is disabled with multiple connections.....	112

Table 5.11 –Average cross-traffic and RPC size of each blob when HTTP Keep-Alive is enabled.....	114
Table 5.12 –Top 5 distinguishing performance metrics for $P0$ with and without HTTP Keep-Alive.	114
Table 6.1 – Description of key metrics collected in Fathom.	125
Table 6.2 – Fathom metrics and their snapshot collection.	126
Table 6.3 – Categorical attributes in Fathom.	128
Table 6.4 – Training features for GMMs. ("tx_latency_*_sec" represents different latency buckets, as described in Table 6.1. In our analysis, we choose 1KB and 8KB.)	131
Table 6.5 – The most distinguishing metrics between blob pairs.....	142
Table 6.6 – Categorical analysis of blobs.	143
Table 6.7 – The 2-Wasserstein distance between blobs	149
Table 6.8 – Categorical Composition of Blobs	153
Table 6.9 – Traffic volume matrix between top pairs of source (row) and destination (column) pods for measurements in the “best” blob (○) and the “worst” blob (○).	157
Table 7.1 – The approach for determining size label for each RPC.	165
Table 7.2 – Cosine similarity between job names that appear in five blobs in different GMMs based on the original and processed job names.	172
Table 7.3 – Blobs matched in the three GMMs based on the Wasserstein distance.	173
Table 7.4 – Cosine similarity of source jobs between GMM blobs in the HDC cluster.	176
Table A.1 – Information collected via SNMP on switches.	187
Table B.1 – A comparison of the average performance with and without HTTP Keep-Alive.	189
Table B.2 – Top 5 distinguishing performance metrics among blobs in each GMMs.	190

Table B.3 – Distance measure of the 2-dimensional Wasserstein distance between blobs in two GMMs.....	190
Table B.4 – Top 5 distinguishing performance metrics for P_0	196

LIST OF FIGURES

Figure 1.1 – An example of Simpson’s paradox: Boxplots of transfer latency overall vs by RPC sizes (Small: [0, 8KB), Medium:[8 KB, 256 KB), Large: [256 KB, ∞)).	8
Figure 3.1 – RPC workflow: Client Side	39
Figure 3.2 – RPC workflow: Server Side	40
Figure 4.1 – Modeling Pipeline	48
Figure 4.2 – Probability density plot of transfer latency in one data center. Values on the x-axis are hidden due to proprietary reasons.	50
Figure 4.3 – Elbow detection for an example BIC curve (x-axis: number of blobs in a GMM; y-axis: BIC score).	53
Figure 4.4 – Comparison of two GMMs with different number of blobs. For goodness of separation, GMM_1 (a) is preferred.	55
Figure 4.5 – Example GMMs. For goodness of interpretation, GMM_2 (b) is preferred.	56
Figure 4.6 – An example of goodness of interpretation analysis with RPC measurements in a production HDC.	60
Figure 4.7 – An example of model selection with a production dataset: the 2D distribution of 95th percentile pacing rate (<i>x-axis</i>) and delivery rate (<i>y-axis</i>) when the number of blobs in GMM is 5 (<i>left</i>), 6 (<i>middle</i>), and 7 (<i>right</i>) respectively (actual values are proprietary and hidden).	61
Figure 4.8 – The results of elbow detection and impurity analysis in example 1. The number of blobs in GMMs ranges from 3 to 17.	62
Figure 4.9 – An example of model selection with a production dataset: the 2D distribution of the median congestion window (<i>x-axis</i>) and the 95th percentile delivery rate (<i>y-axis</i>) when the number of blobs in GMM is 4 (<i>left</i>), 5 (<i>middle</i>), and 6 (<i>right</i>) respectively (actual values are proprietary and hidden).	63
Figure 4.10 – An example of how an outlier distorts the GMM result. The inner small circles illustrate the scatter plots of measurements in a 2D dimension, with the outer circles representing the contour plot of the 2D distribution for measurements in each blob.	64

Figure 4.11 –Time series plot of latency in 95th percentile from Y_1	69
Figure 4.12 –Probability distributions of latency from Y_1 in Figure 4.11.	70
Figure 4.13 –Time series plot of latency in 95th percentile in the modified Yahoo dataset.	70
Figure 4.14 –Probability distribution of latency in the modified Yahoo data set in Figure 4.13.	71
Figure 4.15 –Probability distribution of latency in the modified Yahoo data set in Figure 4.13.	72
Figure 5.1 – Caption for LOF	79
Figure 5.2 – Network topology on CloudLab.	80
Figure 5.3 – Delay information extracted using tcpdump.	82
Figure 5.4 – SACK visualization: cumulative ACK=21 and SACK=31-51,61-71	84
Figure 5.5 – Time series of multiple performance metrics.	88
Figure 5.6 – Kde plots of multiple performance dimensions in the experiment. The unit of the y-axis in a kde plot is probability density. This means that the height of the curve at a particular point on the x-axis represents the probability of a randomly selected data point falling within a very small interval around that point. It is important to note that probability density is not the same as probability. Probability density is a measure of how likely a data point is to fall within a particular range, while probability is a measure of how likely a data point is to have a particular value.	89
Figure 5.7 – Studying the correlation of multiple performance metrics using brushing and linking with 10% of RPCs collected in the experiment.	90
Figure 5.8 – Delay information for each blob in the GMM.	93
Figure 5.9 – Kde plots of maximal client delay, 95th percentile of network delay, throughput, and 95th percentile of bytes-in-flight in each blob based on GMMs.	94
Figure 5.10 –Time sequence graph of one example RPC with a size of 3,171,683 bytes in each blob based on pcap files captured on the server end.	95
Figure 5.11 –Time sequence graph of client delay for the example RPC in Figure 5.10(a) based on pcap files captured on the client end.	97

Figure 5.12 –Scatter plots in throughput and 50th bytes-in-flight with different clustering algorithms. Note that DBSCAN uses the value -1 to indicate outliers that do not belong to any cluster. This suggests that DBSCAN is not effective in this case, as it is unable to cluster all of the data points.....	99
Figure 5.13 –Performance comparison in the 95th percentile network delay, throughput and 95th percentile bytes-in-flight with and without GSO.	102
Figure 5.14 –Cumulative probability of multiple performance metrics of each blob with and without GSO.	103
Figure 5.15 –Delay breakdown in each blob with and without GSO.	105
Figure 5.16 –Cumulative distribution of RPC sizes of each GMM blob when GSO is on.....	106
Figure 5.17 –Cross-traffic information of each GMM blob when GSO is on.....	106
Figure 5.18 –Time sequences of bytes-in-flight and RTT for sample traces in blobs 0 and 2.....	108
Figure 5.19 –TCP time-sequence graph without (left) and with (right) HTTP Keep-Alive. Note that the time in (a) corresponds to a single request, while the time in (b) is for a later request sent after the establishment of a keep-alive connection.	109
Figure 5.20 –Breakdown of delay on end-hosts and network for each blob in GMMs with multiple connections.	110
Figure 5.21 –Kde plots of each blob in throughput and bytes-in-flight when HTTP Keep-Alive is disabled (<i>Disabled</i> : left column) and enabled (<i>Enabled</i> : right column).	112
Figure 5.22 –Number of concurrent RPCs and completion rate with and without HTTP Keep-Alive.	115
Figure 5.23 –Sample time sequence plots of bytes-in-flight and RTTs of example traces in <i>P0</i> and <i>P2</i>	116
Figure 5.24 –Delay breakdown of each GMM blob.	118
Figure 5.25 –The ratio of RPCs from each machine in each blob. "s_" in the machine name indicates servers.	119
Figure 5.26 –The ground truth statistics based on ifconfig and tcpdump.	120
Figure 5.27 –KDE plots of delay, rate and volume without GMMs.	121

Figure 6.1 – Boxplots of the overall transfer latency (left) and receive queueing latency (right) for the application.	132
Figure 6.2 – Boxplots of transfer latency (top left), receive queueing latency (top right), RPC transmit queueing latency (bottom left), and RPC size (bottom right) for the application after aggregating based on RPC sizes.	133
Figure 6.3 – Boxplots of transfer latency (top left), receive queueing latency (top right), app queueing latency (bottom left), and pacing latency (bottom right) for 10 different source jobs in this application.	134
Figure 6.4 – Boxplots of transfer latency, receive queueing latency and TCP queueing latency for RPCs from 200 jobs.	135
Figure 6.5 – Boxplots of transfer latency (left), RPC transmit queueing latency (middle), and RPC size (right) for the application after aggregating based on RPC sizes.	136
Figure 6.6 – <i>StoreService</i> : Time-series of normalized 95th percentiles of minimum RTT, cwnd, and delivery rate	137
Figure 6.7 – GMM Modeling: 95th percentiles of delivery rate, congestion window size, and minimum RTT.	139
Figure 6.8 – Q-Q plot of measurements in each blob (from left to right: <i>O</i> , <i>F</i> and <i>S</i>).	140
Figure 6.9 – Boxplots for the three most distinguishing metrics.	142
Figure 6.10 – Time series plots of the median delivery rate one week before and after deploying <i>newCC</i>	147
Figure 6.11 – 3D scatter plots of RPCs from job <i>J</i> in <i>StoreService</i> in the three weeks before and after the deployment of <i>newCC</i>	148
Figure 6.12 – Boxplots for the most distinguishing metrics.	149
Figure 6.13 – <i>StoreService</i> : 3D scatter plot—before and after	150
Figure 6.14 – Distribution of 5th percentile of cwnd with <i>new_cc</i>	151
Figure 6.15 – <i>MonitorService</i> : GMM blobs	153
Figure 6.16 – Boxplots for the most distinguishing metrics.	153
Figure 6.17 – <i>DatabaseService</i> : Time-series of normalized 95th percentile minimum RTT and 5th percentiles of cwnd and delivery rate across six days.	156

Figure 7.1 – The comparison between the original <i>per-packet delivery rate</i> and the modified <i>per-RPC delivery rate</i> in Fathom.	167
Figure 7.2 – Time series of the normalized 99th percentile (blue), 95th percentile (green), and 50th percentile (red) of transfer latency and delivery rate of the storage service RPCs before and after the deployment of PLB.	168
Figure 7.3 – Latency for small ($\leq 1\text{KB}$) RPCs of storage workload.	169
Figure 7.4 – Latency for large (2MB+) RPCs of storage workload.	169
Figure 7.5 – 2D distribution of each GMM blob for RPCs in MLService from two three clusters (x-axis: normalized delivery rate, y-axis: normalized transfer latency).	173
Figure 7.6 – Hierarchical clustering dendrogram of the GMM blobs in Figure 7.5.	174
Figure 7.7 – Distance between GMM blobs and the breakdown of average latency in each blob for large RPCs.	177
Figure 7.8 – Distance between GMM blobs and the breakdown of average latency in each blob for medium-sized RPCs.	178
Figure 7.9 – The breakdown of average latency in each GMM blob for small-sized RPCs.	179
Figure 8.1 – Analysis Pipelines Adopted in Key Prior Work	182
Figure 8.2 – A General Network Monitoring Framework—GMM analysis informing network health monitoring and planning, general rollout impact analysis, anomaly detection, and anomaly classification.	183
Figure A.1 – Example output of ifconfig	187
Figure A.2 – An example trace captured by tcpdump.	188
Figure B.1 – Breakdown of delay on end-hosts and network for each blob with and without HTTP Keep-Alive in the GMMs with a single connection.	190
Figure B.2 – Kernel density plots of throughput, bytes-in-flight, and network delay for multiple connections when HTTP Keep-Alive is disabled (<i>Disabled</i> : left column) and enabled (<i>Enabled</i> : right column).	192
Figure B.3 – Time sequence plots of bytes-in-flight and round-trip times (RTTs) of selected traces from blobs <i>P0</i> and <i>P2</i> using multiple connections.	194

Figure B.4 – Ground truth information from pcap files.....	194
Figure B.5 – TCP tuple information from one server during experiments with different communication patterns.	195
Figure B.6 – Delay breakdown for blobs in each GMM with different communication patterns.	196
Figure B.7 – KDE plots of P_0 with different communication patterns: 1 to 1 vs. 1 to 5.....	197
Figure B.8 – The ratio of RPCs in each blob with different communication patterns.	197

CHAPTER 1: INTRODUCTION

Hyperscale Data Centers (HDCs) (1) are among the largest distributed computing machines ever constructed. These HDCs, such as Amazon AWS (2), Microsoft Azure (3), and Google Cloud (4) contain tens of thousands of servers deployed in each of more than a dozen mega-scale data centers throughout the world (5; 6). HDCs simultaneously host thousands of applications that provide different types of services for a wide range of purposes, including education (e.g., Google translation and Zoom), travel (e.g., Airbnb, Google Maps), finance (e.g., Bank of America, Capital One), shopping (e.g., Amazon, Walmart), and entertainment (e.g., Youtube, Facebook). Most HDC applications are business-critical and performance-sensitive, serving millions of enterprise customers and generating billions of dollars in revenue.

As the demand for new applications such as artificial intelligence, Internet-of-Things (IoT), and game streaming (e.g., Stadia (7)) continues to rise, the workload in HDCs has been increasing rapidly. As a result, the network bandwidth requirements in HDCs have roughly doubled every 9 months in recent years, outpacing other infrastructure components (8). Due to the significant dependence of HDC applications on the network infrastructure, it is essential to monitor the network performance of HDC applications.

1.1 Understanding the Network Performance of HDC Applications is Critical

1.1.1 The Significance of Understanding the Network Performance of HDC Applications

The performance received from the underlying network infrastructure is a major determinant for the overall performance of HDC applications. The importance of the network infrastructure in HDCs is highlighted by the fact that network bandwidth requirements have almost doubled every 9 months for the past several years, a rate of growth that surpasses that of other infrastructure

components (8). A thorough understanding of HDC application network performance is important for performing three fundamental tasks: assessment, planning, and diagnosis. As discussed below, these three tasks not only help to evaluate the health of HDCs and applications running inside them, but also ensure satisfactory performance in the long term.

1.1.1.1 Task 1: Assessment

Assessment helps to determine whether HDC applications are performing optimally given existing HDC infrastructure. If not, engineers can identify solutions to better utilize the HDC infrastructure and improve the network performance of HDC applications. For example, when assessing the performance of applications that primarily generate small transfers (e.g., monitoring services), we can compare their network latency with the propagation delay calculated based on the maximum physical distance between the end hosts. If the latency is close to the propagation delay, the applications may be performing at their best. However, if the latency is significantly larger, it could indicate that there is a high volume of cross-traffic along the network path, which is taking up a large share of network resources and introducing a noticeable queuing delay. In this case, the applications may be able to improve their performance by scheduling their application processes in different locations to reduce bandwidth sharing, or advocating for a better load balancing policy.¹ Similarly, when assessing the performance of applications with bulk transfers (e.g., streaming services), comparing their transmission rate with the available bottleneck bandwidth on the network path can reveal whether they are fully utilizing the available resources. It is important to have a good understanding of the network performance of applications in order to conduct a useful assessment.

¹Distributed application processes are the processes that run on the different computers that make up a distributed application. These processes communicate with each other to coordinate the execution of the application.

1.1.1.2 Task 2: Planning

Planning ensures that HDC infrastructures continue to meet the requirements of evolving service and customer demands in a cost-effective way, as newer applications emerge and traffic workloads increase. Good planning is important for avoiding revenue loss, reduced productivity, or unacceptable customer experience. A thorough understanding of application's network performance is valuable in this decision-making process because it can help identify which parts of the infrastructure will significantly improve performance if upgraded, or reduce operating costs without degrading performance if removed. For example, if HDC applications are not typically bandwidth limited (e.g., they are CPU limited), operators can reduce the bandwidth to reduce costs with little impact on application performance. On the other hand, if network operators anticipate increasing traffic demands, they may consider deploying more bandwidth resources to meet those demands.

1.1.1.3 Task 3: Diagnosis

Diagnosis involves identifying the underlying causes and finding remedies for performance-related anomalies. Performance anomalies are common in HDCs. For example, Microsoft Azure reports thousands of virtual machine-down events daily (9). There are several potential causes of performance anomalies, such as buggy software, hardware, or network configurations. Each of these issues may require a different solution from a different engineering team. Therefore, it is important to first identify the component(s) responsible for abnormal performance. For example, is the issue due to insufficient computing power on local hosts, or is it caused by bugs in the application code? In the former case, distributing computations to more machines with better configurations may alleviate the issue.

Having a good understanding of application performance can significantly assist engineers in resolving performance-related issues in at least two ways. First, it can help to reveal where anomalies occur. For example, do the anomalies affect traffic from a single application, or do they span multiple applications? Are these applications running in the same locations in HDCs,

possibly sharing switches or links, in which case the source of the anomaly can be localized?

Second, it can help to understand why anomalies occur. For instance, knowing that all anomalous traffic experiences increased latency on remote hosts, rather than over the network, can point the investigation towards resources on remote ends (e.g., CPU, memory, storage). This information can help narrow the scope of the investigation and can accelerate the debugging process.

In conclusion, a comprehensive understanding of the network performance of applications in HDCs is essential for network operators and engineers to efficiently assess the health of the HDC infrastructure and applications, plan for future evolutions and upgrades, and diagnose performance anomalies. However, building a comprehensive understanding of HDC application network performance is challenging due to their characteristics described next.

1.1.2 Common Characteristics of HDC Applications

Unlike single-enterprise data centers and supercomputers, which concurrently run a relatively few applications, HDCs host thousands of applications simultaneously. These applications may differ in service types and customer bases, but they share three common characteristics: They are *distributed*, *diverse*, and *coupled*.

1.1.2.1 Distributed

HDC applications are designed to be distributed for scalability and flexibility. Each application may consist of tens to thousands of distributed application processes that communicate and coordinate through multi-Gbps links of multi-Tbps network configurations, consuming and producing terabytes of storage data in the process (10; 11). The overall performance of these distributed processes is highly dependent on the performance of their network transfers from the underlying infrastructure. There are three main entities that influence the behavior of network transfers: the local host, the remote host, and the network. The local host generates the data to be sent and initiates the communication, the remote host receives and processes the data, and the network transmits the data between the local and remote hosts. For transferring data between

local and remote hosts, Remote Procedure Call (RPC) is a fundamental protocol and is used by most applications in HDCs. It allows an application program to call a procedure regardless of whether it is local or remote. RPC uses the request/reply paradigm, in which a client sends a request message to a server and the server responds with a reply message, with the client suspending execution while waiting for the reply. In this thesis, the term RPC is used to represent both actual RPC protocols (e.g., gRPC (12)) and generic request/response protocols (e.g., HTTP (13)).

For understanding the overall performance of distributed application processes, it is essential to understand their performance at each of the three stages. For instance, how quickly can a 1 kilobyte packet be generated on the local host? How long does it take for the packet to traverse the network and reach the remote host? How long does the packet remain in the receive buffer on the remote host before being processed?

1.1.2.2 Diverse Application Workload & Requirements

HDC applications exhibit significant diversity. They range from *production* services such as video streaming, social networks, email, search, and enterprise cloud services (14; 15; 16; 17; 2; 18), to *support* services such as storage, computation, and performance monitoring services (with multiple versions of each service to support different requirements for access latency, resolution, accuracy, and priorities) (19; 20; 21). These applications have vastly different traffic workloads, resulting in diverse network performance. The emergence of new technologies, such as artificial intelligence, Internet-of-Things (IoTs), and game streaming (e.g., Stadia (7)), continually adds to the diversity of applications and workloads in HDCs. Additionally, their performance may vary geographically and temporally (daytime vs. evening on different continents). HDC applications also differ in their performance requirements for latency, bandwidth, and loss (22; 23). For example, interactive services (e.g., panning a map) typically have a latency requirement of 100 ms or less (24), while less interactive ones (e.g., emails, downloading documents) may have an acceptable latency range of 1-16 seconds (25; 26; 27).

1.1.2.3 Diverse Infrastructures

HDC infrastructures can also be fairly diverse. HDCs are structured hierarchically for scalability and efficiency, with servers hosted in *racks* that are interconnected through top-of-rack (ToR) switches. Multiple *racks* are grouped into *Pods*, and multiple *Pods* are grouped into *clusters*. Each *Pod* contains only servers in one *cluster*, while a *cluster* may span several *Pods*. Multiple *clusters* are interconnected within a *metro* region. Each level of this hierarchy may have different hardware (CPU, memory, and disk), software (kernel versions, libraries), and network configurations (bandwidth, network interface controller (NIC), and transport protocols). For instance, switches can have different NIC speeds, ranging from 10 Gbps to 200 Gbps, and buffer sizes, varying from 6 MB to 12 MB. These differences can impact the serialization and queueing latency experienced by applications in the network. For a comprehensive overview of a production datacenter infrastructure and its evolution over time, please see references (28) and (29).

In addition, HDC infrastructure is continually being re-engineered for improving performance and meeting increasing demands. Given the scale of HDCs, it can take months to years to complete a global software/hardware rollout (30; 31), and multiple versions can co-exist in different parts of an HDC. The diverse network infrastructures can lead to different network performance even for the same application. All of these factors contribute to significant variations in application network performance, making it difficult to model using a small number of application behavioral models or a uniformly-applicable set of heuristics (32).

1.1.2.4 Coupled

The performance of different HDC applications is coupled. HDC applications share infrastructure resources, such as computing, network, and storage resources, which are managed dynamically based on demand and controlled by operator-specified policies. Competition for these shared resources creates a complex coupling between applications and adds a significant non-determinism in their performance. Additionally, HDC applications rely on each other. For example, a web application A that allows users to create and manager their profiles relies on services

provided by a database application B for storing data, which may in turn rely on a web server C that host application B. Therefore, failures in application C also degrades the performance of applications A and B. In summary, a poorly performing application can have a cascading effect on multiple other applications, which may be difficult to model due to complex interdependencies (33; 34; 35).

Furthermore, the usage of HDC resources is coupled. For example, increasing network bandwidth resources may put additional pressure on local CPU and memory resources, since these will need to handle a larger volume of requests and responses. Similarly, an uneven distribution of workload across storage resources can create hotspots in the network and cause severe congestion on specific links and switches. These couplings can make it a lengthy and challenging task to understand the performance of applications, requiring close cooperation between teams that manage different services and resources (36; 33).

1.1.3 Challenges in Understanding the Network Performance of HDC Applications

Monitoring and understanding the network performance of HDC applications can be challenging due to the enormous scale, tremendous diversity, and complex couplings. With the number of applications at scale, HDCs can generate tens of terabytes of performance data per day. Therefore, it is impractical to manually analyze the performance of each network transfer for each application worker. Analyzing the behavior of network transfers from a few applications provides only partial information and can lead to biased conclusions, as performance can vary significantly across HDC applications. Additionally, performance anomalies can occur unexpectedly at any time and place, making it critical but challenging to promptly identify and troubleshoot them due to the massive scale.

In addition, the numerous variations in HDCs may result in a partial or even misleading understanding of applications' network performance due to Simpson's paradox (37; 38). Simpson's paradox suggests that a trend visible in multiple groups may be hidden, or even reversed, when those groups are aggregated. To illustrate this pitfall, Figure 1.1 shows the distributions of trans-

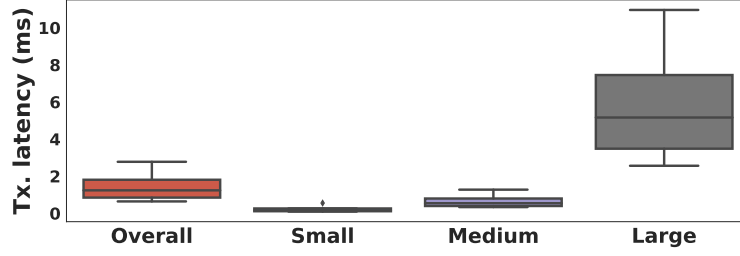


Figure 1.1: An example of Simpson’s paradox: Boxplots of transfer latency overall vs by RPC sizes (Small: [0, 8KB), Medium:[8 KB, 256 KB), Large: [256 KB, ∞)).

fer latency for a storage service in a production HDC. Transfer latency is the interval from when the RPC data hits the wire until it is fully acknowledged in TCP. Transfer latency increases with RPC size, with the highest latency for large RPCs. However, as can be seen from the overall distribution, it is difficult to tell how transfer latency varies across different-sized RPCs. In HDCs, a common approach for scalable performance monitoring is to focus on aggregated performance metrics from large groups of application instances, such as the 95th/99th percentile of transfer latency over a 1-minute interval (19). However, in a diverse HDC environment with different applications characterized by different volumes of traffic, service types, and infrastructures, it is important to carefully aggregate performance metrics to avoid Simpson’s paradox (37; 38). Aggregating traffic based on heuristic rules derived from external information, such as application names, job names, or locations, may not be effective because even traffic generated by the same application may experience different performance based on factors such as time, location, configuration, customer demand, and service type. Therefore, finding an aggregation approach that reduces the analysis and storage overhead while avoiding Simpson’s paradox is a critical but challenging task.

Finally, the performance metrics of HDC applications can be independent or coupled depending on the underlying HDC infrastructure. There are three main types of metrics that collectively describe an application’s network performance: the rate (how fast traffic can be delivered), the latency (how long it takes to deliver the traffic in different stages), and the volume (how much traffic is being transferred). For instance, the delivery rate can be improved by increasing the sending rate without affecting end-to-end latency, but only up to a limit that is defined as bandwidth-

delay product and is calculated as the product of a link's capacity and the round-trip delay time. Bandwidth-delay product represents the maximum amount of data on the network circuit at any given time, i.e., data that has been transmitted but not yet acknowledged. Beyond the limit, the delivery rate stays constant and latency increases. Therefore, when monitoring and analyzing application performance, it is important to consider multiple performance metrics simultaneously because each of them provides only partial information. In production HDCs, consequently, engineers may need to examine tens of performance metrics manually for each of thousands of applications, which can be both tedious and prone to errors.

In summary, HDC applications are distributed, diverse, and coupled. A thorough understanding of their performance in the underlying HDC infrastructure is crucial for facilitating assessment, planning, and diagnosis tasks. These tasks are essential for evaluating the health of HDCs and the applications running within them, as well as for ensuring satisfactory long-term performance. However, the large scale of HDCs makes it difficult to analyze performance metrics with manageable overhead. The diversity and coupling in HDCs also complicate analysis due to Simpson's paradox and inherent non-determinism.

1.2 Performance Monitoring in HDCs: State-Of-The-Art

In recent decades, researchers have made significant efforts to understand the performance of applications and different components in HDCs. Previous studies have focused mainly on three directions: performance monitoring/tracing, anomaly detection, and anomaly characterization. Performance monitoring/tracing develops tracing systems for collecting detailed performance data covering different components in HDCs with manageable overhead (39; 20; 19; 21). Despite the availability of performance data in HDCs, extracting interpretable information remains a challenge due to the scale, diversity, and coupling. For analyzing performance data in HDCs, prior research has focused mainly on two tasks: anomaly detection and anomaly characterization (19; 40; 21; 35; 41; 31). Anomaly detection aims to promptly flag anomalous performance, that is, when and where anomalies occur, while anomaly characterization focuses on classifying and

localizing anomalous behavior, that is, what network component is most likely to be responsible. The challenges of scale and complexity are addressed mainly using learning-based techniques. For example, advanced deep neural networks, such as convolutional neural networks (CNN (42)), recurrent neural networks (RNN (43)), and long short-term memory networks (LSTM (44)), have been used to capture complex temporal dependence across multivariate time series performance data (45; 46; 47; 35). A detailed summary of related work is represented in Chapter 2.

While these approaches are successful in detecting when a performance anomaly occurs and even localizing it, they cannot explain why or how an anomaly occurs and how it should be resolved due to the black-box nature of learning-based algorithms. Furthermore, these approaches do not provide interpretable insight into the normal performance of applications, which is necessary for general assessment and planning tasks.

1.3 Our Approach: Analyzing HDC Application Network Performance Via Constraints-based Models

1.3.1 Our Goal

In this thesis, our objective is to understand the network performance of HDC applications by identifying the constraints that limit their performance. Specifically, we propose a constraint-based approach for modeling the network performance of HDC applications using Gaussian Mixture Models (GMMs) (48). This approach enables us to achieve two objectives:

- **Interpretability** HDC applications are designed to run as fast as possible by using all available resources (49; 50). As a result, the performance of their application processes is always determined by constraints that bound “possible”. By understanding the constraints that limit their performance, we can make informed decisions on how to improve it, such as increasing or decreasing the network bandwidth.
- **Dimension Reduction** Analyzing application network performance in HDCs is challenging due to the enormous volume of performance data generated. This data includes

hundreds of metrics relating to billions of application processes from thousands of different types of applications running on diverse infrastructures. The scale of the data can be overwhelming, making it difficult to analyze and interpret. However, despite the large number of metrics, there are only a few possible constraints that may limit the network performance of application processes. Furthermore, groups of application processes performing similar tasks in similar parts of the infrastructure are likely to encounter similar constraints. By focusing on these constraints instead of individual performance metrics, we can effectively characterize the network performance of a large number of processes.

1.3.2 Implementation

To achieve our objective, we have developed a pipeline for modeling and analyzing the network performance of HDC applications using GMMs. The pipeline consists of two main components: GMM modeling and GMM analysis. The GMM modeling component takes RPC performance data as input and generates a GMM that accurately characterizes different observed behavior in RPC network performance. By considering multiple performance dimensions and their correlations, it groups RPCs with similar performance into a *blob* and separates these with divergent performance into different *blobs*. The GMM analysis component employs statistical techniques to analyze the results generated by the modeling pipeline. It extracts meaningful information from the GMMs, which can be used to understand different performance experienced by RPCs in each blob and to inform future actions.

The pipeline has been implemented in one of the largest production HDCs at Google, and has been used for modeling the network performance of real-world applications for the past three years.

1.3.3 GMM Evaluation

We evaluate the effectiveness of GMM-based analysis through several controlled experiments on the Cloudlab emulation platform (51) and real-world applications in the production HDC.

On Cloudlab, we employ a hierarchical network topology (see Figure 5.2 in Section 5 for details) and use traffic patterns that follow the distribution of a representative application in the Facebook data center (52). Performance data is collected from both switches and end hosts using Simple Network Management Protocol (SNMP (53)), ifconfig (54), and tcpdump (55). The performance data is either used as input for modeling GMMs or as ground-truth information for validating GMM results. Through several controlled experiments, we demonstrate that GMMs are capable of identifying different types of performance experienced by RPCs, evaluating the impact of different configuration factors on the network performance of RPCs, and detecting performance anomalies and pinpointing their most likely causes.

In the production HDC, we use an existing RPC instrumentation system (Fathom) developed in the production HDC for collecting RPC telemetry. Through several case studies, we demonstrate the efficiency of our modeling approach in the following areas:

- Evaluating the performance of a major storage infrastructure service. This service accounts for nearly 20% of the entire traffic inside the datacenter, which runs multiple jobs, communicates with several other clients and backend servers, and transmits over multiple transmission priorities (QoS classes). Our GMM analysis revealed how QoS classes and RPC sizes leads to different network performance experienced by the service.
- Understanding the impact of a Transmission Control Protocol (TCP) congestion control algorithm (Section 6.4.1) change (cc_{old} vs. cc_{new}) on application network performance. We used RPC measurements from over 80 million sampled RPCs collected three weeks before and three weeks after the change. Our GMM-analysis suggested that cc_{old} allowed RPCs with higher-priority QoS classes to occupy a larger share of bandwidth resources, leading to significant performance gaps among RPCs with different QoS classes. By comparing the GMMs before and after the change, we verified that cc_{new} effectively reduces the performance gap. Besides, it reduced packet losses and queueing latency for RPCs across all QoS classes.

- **Planning for future infrastructure upgrades.** We evaluated the network performance of a global monitoring service that operates on a datacenter with heterogeneous switch-port speeds. Our GMM analysis, which used RPC measurements aggregated over a million sampled RPCs, revealed that a particular subset of RPCs would see performance improvements from increased bandwidth capacity, as their performance was still limited by bandwidth even with fast-speed ports. On the other hand, a distinct set of RPCs was primarily constrained by host resources. Therefore, increasing the network bandwidth would not improve their performance, but upgrading the processing resources on the end hosts could be more beneficial.
- **Troubleshooting performance degradation.** By modeling RPC measurements aggregated over 40 million sampled RPCs after the start of a performance outage, GMM identified a GMM blob experiencing significantly poor network performance due to more severe network constraints, which are hard to recognize from the time-series performance data. GMM analysis revealed that these RPCs were heavily skewed toward communications between two datacenter locations. Further analysis confirmed that these two locations were improperly configured and had insufficient bandwidth for the amount of storage deployed inside.

1.3.4 Enhancements

We identified two potential issues with Fathom that could affect the accuracy of GMM modeling: Simpson’s paradox in performance data caused by the disproportionate ratio of small and large RPCs, and the volatility of per-packet delivery rate. To address these issues, we introduced an additional categorical attribute in Fathom to describe RPC size and modified the way Fathom collects delivery rate. After implementing these modifications, we present two additional case studies. The first case study illustrates how GMMs can be used to assess the impact of an engineering change in load balancing on application performance in the HDC (56). The second case study assesses the network performance of a machine learning service across the fleet of

HDCs. To gain deeper insights from the GMM results, we improve the GMM analysis pipeline in two ways. Firstly, we preprocess the job names associated with the ML service using Time Frequency-Inverse Document Frequency (TF-IDF (57)). This enables us to uncover distinct job information associated with RPCs in different GMM blobs. Secondly, we use hierarchical agglomerative clustering (58) to group GMM blobs based on their modeling results. This approach helps us to efficiently identify blobs that experience different performance across the fleet with diverse infrastructure, and facilitates further analysis to understand the underlying causes of the performance variations.

Through extensive controlled experiments on the Cloudlab and several case studies based on real-world applications in the production HDC, we have demonstrated the efficacy of our constraint-based approach in modeling the network performance of HDC applications.

1.4 Thesis Organization

The thesis is organized as follows: Chapter 2 summarizes the approaches and limitations of prior work in performance monitoring/tracing, anomaly detection, and anomaly characterization; Chapter 3 introduces our approach for modeling the network performance of applications in HDCs based on their bottleneck constraints using GMMs; Chapter 4 illustrates the detailed implementations of our GMM modeling and GMM analysis pipeline; Chapter 5 evaluates the effectiveness of using GMMs on the Cloudlab testbed with controlled experiments; Chapter 6 implements the pipeline in a production HDC and demonstrates its efficiency with real-world applications through several case studies, Chapter 7 summarizes the modifications made to further improve the pipeline in the production HDC and presents additional case studies, and Chapter 8 discusses limitations and future directions of this work.

CHAPTER 2: RELATED WORK

For many years, researchers have been working to improve the resource utilization and performance of HDCs by studying the performance of their different components. Prior work in this area has focused on three main topics: performance monitoring/tracing, anomaly detection, and anomaly characterization. **Performance monitoring/tracing** provides an overview of how different components in HDCs perform by collecting fine-grained performance data with manageable overhead. **Anomaly detection** analyzes performance data for flagging anomalous HDC performance that deviates from historical behavior or expectations based on domain knowledge. **Anomaly characterization** focuses on locating the root causes of the identified anomalous performance.

This chapter summarizes prior work in these three areas and elaborates on how these are addressed in this dissertation. Section 2.1 introduces distributed monitoring/tracing systems developed for HDCs. Section 2.2 discusses different techniques for anomaly detection and characterization, as most research in this field aims to address both goals jointly.

2.1 Performance Monitoring/Tracing

To understand the performance of HDC components and the applications running on top of these, large-scale monitoring and tracing systems have been developed for continually collecting fine-grained performance data for further analysis (59; 39; 60; 61; 62; 20; 19). A key focus of these systems is achieving complete coverage with manageable overhead given the enormous scale and the sheer amount of traffic in data centers. Prior work includes systems that log major events within monitored distributed calls (e.g., Remote Procedure Calls), systems that perform

packet-level network telemetry, and systems that proactively launch low-overhead probes to measure latency. Below, we describe some representative systems deployed in production HDCs.

Modern HDC applications are typically implemented in a distributed manner. These are constructed from collections of software modules that may run on thousands of machines located in multiple physical facilities. Tracing the performance of such distributed systems is essential for understanding behavior and performance issues. Sigelman et al. (39) designed *Dapper*, a large-scale distributed system tracing infrastructure at Google. Given the size of Google’s services and data centers, *Dapper* is designed to meet three requirements: low overhead, application-level transparency, and scalability. For low overhead and scalability, *Dapper* applies adaptive sampling and demonstrates that sampling less than 0.1% of requests provides sufficient information for many tracing purposes, including tracking application performance, identifying performance issues, and testing impact of changes. For application-level transparency, *Dapper* restricts its core tracing instrumentation to a sufficiently low level in the software stack Remote Procedure Call (RPC) library, including a small corpus of ubiquitous threading, control flow, and RPC library code. Therefore, no additional annotations are needed to trace even large-scale distributed systems such as Google web search. *Dapper* logs start times, end times, and RPC timing information (such as Client Send, Client Recv, Server Recv, and Server Send) associated with each sampled RPC. Since being implemented at Google, *Dapper* has been quite useful for developers and operations teams and has evolved from a self-contained tracing tool into a monitoring platform that enables the creation of many analysis tools. Similar to *Dapper*, Twitter designed *Zipkin* (62), Uber introduced *Jaeger* (61), and Facebook built *Canopy* (63) for distributed tracing. More recently, Las-Casas et al. (64) proposed *Sifter* to sample traces in a biased approach that weighs how “interesting” their content is to catch edge-traces, infrequent request types, and anomalous executions. Compared to redundant, pervasive common-case execution traces collected with uniform random sampling, the corner cases presented above are often more useful for analysis and troubleshooting tasks. *Sifter* assigns each event a unique label based on the origin of the event and models the conditional probability of a label occurring given its immediate causal

predecessors and successors with a neural network. It then identifies uncommon traces based on prediction loss. *Sifter* has been demonstrated to bias effectively towards anomalous and outlier executions after being integrated into several open-source tracing systems.

Ren et al. (60) designed a continuous profiling infrastructure called Google Wide Profiling (GWP) to provide performance insights of machine resources consumed by cloud applications at Google. GWP samples across machines in multiple data centers and collects events such as stack traces, hardware events, lock contention profiles, heap profiles, and kernel events. It allows cross-correlation with job scheduling data, application-specific data, and other information. GWP collects both whole-machine and per-process profiles. Whole-machine profiles capture all activities on the machine, such as user applications, the kernel, kernel modules, daemons, and other background jobs, and include hardware performance monitoring (HPM) event profiles, kernel event traces, and power measurements. Per-process profiles include heap allocation, lock contention, wall time, CPU time, and other performance metrics. GWP shows how cloud applications consume machine resources over time and helps developers design, evaluate, and calibrate their applications.

Zhu et al. (20) presented *Everflow*, a packet-level network telemetry system. For each packet trace, *Everflow* maintains one copy of the full packet content and a set of per-hop information, including the IP address of the switch where the packet is mirrored, the timestamp, Time to Live (TTL), the source MAC address, and the explicit congestion notification (ECN) (65). This information can help to remedy common faults in data centers, including packet drops, loops in routing paths, inflated end-to-end latency, load imbalance, and protocol bugs. Considering that packet drops can happen for several reasons, such as software bugs or faulty hardware on switches, passive tracing alone may not be sufficient to identify the problem. Therefore, *Everflow* also supports guided probing to replay a packet trace by injecting test packets into target switches and tracing the behavior of the injected packets to help identify the culprit. *Everflow* is demonstrated to be effective after running for 6 months in Microsoft's data centers.

Guo et al. (19) developed *Pingmesh* for network latency measurement and analysis. Network latency between servers in HDCs is a critical performance metric. It can be used to (1) determine if an application’s perceived latency issue is caused by the network or not, (2) define and track network service level agreement (SLA), and (3) troubleshoot network issues automatically. *Pingmesh* is designed to be always on and provide network latency data for all servers. It leverages all servers to launch TCP or HTTP pings and records the round-trip time (RTT) between each of these.¹ To achieve as large coverage as possible, *Pingmesh* maintains a complete graph at three different levels: (1) within a data center *pod*, where all servers under the same ToR (top-of-rack) switch form a complete graph; (2) intra-data center, where each ToR switch is treated as a virtual node and all ToR switches form a complete graph; and (3) inter-data center, where each data center acts as a virtual node and all data centers form a complete graph. *Pingmesh* has been running in Microsoft’s data centers for more than four years. It helps to better identify network latency and packet drops, define and track the network service level agreement (SLA), and determine whether or not a live-site incident occurs because of network issues.

Several monitoring systems have been developed to track resource utilization or application performance in HDCs from different perspectives, and aid in the design of efficient techniques for managing the tracing and storage overhead required by the traffic volume in data centers. Although all of them reveal critical performance information, some are not adequate for comprehensively evaluating the performance of applications in HDCs. For example, GWP only collects machine resources consumed by applications and does not cover their performance over the network, while *Pingmesh* and *Everflow* only indicate what applications may experience over the network, but not on the end hosts. The overall performance of HDC applications is determined by their experience throughout the entire operation. Thus, we have to collect performance data from the local host (CPU, memory, and application data generation), the remote peer (CPU, memory, and application data consumption), and the network that is used to communicate (links, buffers, processing, and switching fabric). Moreover, working with packet traces alone is inadequate be-

¹RTT is defined as the time interval from the time a message is sent from one server to the time it receives the acknowledgment from the receiver.

cause that cannot help differentiate the performance of one application from another. One point where metrics associated with both end hosts and network can be observed is the application's Remote Procedure Call (RPC) library (discussed in detail in Chapter 3). Since the vast majority of applications in HDCs use RPCs, tracing systems that are based on RPCs, such as *Dapper*, are better suited for monitoring the performance of HDC applications.

2.2 Anomaly Detection and Characterization

Performance anomalies are common in HDCs and can occur anywhere, anytime, unexpectedly, and with severe impact. For instance, Microsoft Azure reports on the order of thousands of VM down events daily (9). CBS All Access crashed during the Super Bowl due to increased workloads (66). A failure in Amazon's AWS service lasted hours in November 2020 and had a cascading impact on the performance of many applications, services, and websites, including Autodesk, Capital Gazette, Roku, Glassdoor, and The Washington Post (67). Thus, detecting or even predicting when and where anomalous performance may occur is essential for preventing massive performance degradation given the cascading effects.

Although comprehensive high-resolution performance data are available through monitoring/-tracing systems (Section 2.1), the sheer amount of data in data centers makes it quite difficult to detect and analyze abnormal performance. Additionally, performance anomalies can occur for several reasons, including software failures, hardware failures, and misconfigurations (33; 36). Different types of anomalies, such as hardware misconfigurations, software issues, and network congestion, can affect the performance of applications in different ways in HDCs. These anomalies are typically assigned to different teams (server, client, or network) for resolution. Indeed, network engineers may encounter millions of "alerts" about performance anomalies each day (9) and may have to spend days or weeks investigating just a small selection from these. Diagnosing issues can be a time-consuming and complex process that involves reviewing history logs and data from multiple sources, forming hypotheses, collaborating with different teams, and possibly going through multiple rounds of finger-pointing before the responsible components are identi-

fied (40; 36; 33). This approach is neither scalable nor sustainable. Characterizing the anomalous performance in order to understand the cause and potential solutions is crucial for effective debugging and troubleshooting.

Prior work that addresses these challenges can be classified into two main categories: anomaly detection and anomaly characterization. Anomaly detection relies mainly on representative historical data collected over a long period of time or heuristic rules developed by domain experts for pointing out “anomalous” performance that deviates from past experience or violates the rules. In anomaly detection, one direction pursued in the past has been change-point detection based on high-resolution time series data (68; 69; 70; 71; 72). Most recently, advanced deep neural networks have been used to capture complex temporal and spatial dependence of multivariate time-series performance data in HDCs (Section 2.2.1). Others have considered performance statistics and/or traffic features and applied heuristic rules, co-clustering, or learning-based techniques for distinguishing “normal” from “anomalous” (Section 2.2.2). Existing anomaly detection approaches are accurate and efficient in detecting abnormalities based on historical data. However, they are inherently limited in interpretability because most are black-box techniques that can only point out differences. For resolving anomalies, it is critical to also describe what applications experience in HDCs, reveal the bottlenecks in resources/components that limit performance, and provide hints about the causes of anomalies.

Anomaly characterization focuses on identifying common symptoms (e.g., high latency, significant packet losses), responsible entities (e.g., CPU, memory, network), or shared attributes (e.g., routing links/switches, geographical locations, applications) among the detected performance anomalies to facilitate the debugging process. To this end, some efforts characterize the type/location of factors responsible for an anomaly. For instance, some consider large sets of anomalous events and aim at identifying components in HDCs that should be responsible for a large fraction of anomalies, such as links, switches, virtual machines, or applications/micro-services. Others explicitly model dependency/connectivity between components in HDCs and

apply probability propagation or inference analysis to identify root cause components that explain most of the anomalies.

Next, we discuss in more detail related work on anomaly detection and characterization in HDCs.

2.2.1 Time-series Based Approaches

One extensively studied approach for anomaly detection has been the detection of abrupt changes in time-series data, that is, change-point detection (68; 69; 70; 71). For time-series performance data, the idea is to model stable behavior from the past as normal performance and flag deviations as abnormal. With advances in machine learning in recent years, both supervised (73; 74; 75; 76; 77) and unsupervised approaches (78; 79; 70; 45; 46; 47) have been explored to achieve this goal. In this section, we discuss three recent anomaly detection studies built upon advanced neural networks, which have great competence in uncovering temporal and/or spatial dependence among multidimensional performance data over a long period. Reference (71) provides an extensive survey in this field.

Su et al. (46) proposed *OmniAnomaly*, a stochastic recurrent neural network (RNN) that models temporal dependencies between multivariate time-series data for anomaly detection. For multivariate time-series performance data, *OmniAnomaly* follows three steps: (1) capture the normal patterns of multivariate time-series, (2) reconstruct the input data from their representations, and (3) use the reconstruction probability to determine anomalies. It encompasses three key techniques: Gated Recurrent Units (GRU, a variant of RNN), planar Normalizing Flows (NF), and stochastic variable connection to learn robust latent representations that encode both the temporal dependence and stochasticity of multivariate time-series. Finally, it applies an adjusted Peaks-Over-Threshold method to fit the tail portion of a probability distribution with a generalized Pareto distribution in order to automatically learn the threshold of anomaly scores. To interpret the detected anomalies, *OmniAnomaly* annotates each anomaly with the top few univariate time series ranked by their reconstruction probabilities, which may provide sufficient clues

for troubleshooting. *OmniAnomaly* has been evaluated in three data sets. Compared to existing approaches that fail to capture temporal correlations between multivariate performance data, it achieves better precision, recall, and F1 score.

Zhang et al. (47) introduced a Multi-Scale Convolutional Recurrent Encoder-Decoder (MSCRED) for anomaly detection and diagnosis with multivariate time-series data. MSCRED aims to jointly resolve three tasks: anomaly detection based on reconstruction probabilities, root cause identification according to the anomaly score of each time series, and interpretation of the severity of the anomaly based on its duration. It first constructs multiscale (resolution) signature matrices for characterizing multiple levels of system statuses in different time steps. Given the signature matrices, a convolutional encoder is used to encode correlations between multiple time series, and an attention-based Convolutional Long-Short-Term Memory (ConvLSTM) network is developed to capture the temporal pattern. Finally, a convolutional decoder is used to reconstruct the input signature matrices, and the residual signature matrices are used for detecting and diagnosing anomalies. In extensive empirical studies, MSCRED achieved better precision, recall, and F1-score than state-of-the-art baseline methods on a synthetic data set and a power plant data set. The improvements ranged from 13.3% to 30.0%, depending on the data set and settings..

Ren et al. (45) developed an anomaly detection pipeline that continuously monitors time series data and alerts potential incidents. The algorithm, called SR-CNN, combines a Spectral Residual (SR) model with a Convolutional Neural Network (CNN) by applying the CNN directly to the output of the SR model. The SR model uses Fourier transforms to identify anomalies in the original time series data, while the CNN develops a more sophisticated method for detecting performance anomalies by replacing the single threshold used by the original SR solution. To train the discriminator, synthetic data is generated by randomly selecting points in the time series, calculating the injection value to replace the original point, and obtaining its saliency map. The algorithm was evaluated on two public datasets and an internal production dataset at Microsoft. It consistently demonstrated better accuracy, efficiency, and generalizability than the state-of-the-art approaches. For example, on the Microsoft dataset, the F1-score increased from 0.443 to 0.537.

The above techniques for flagging and ranking anomalous metrics can be applied to multi-dimensional time-series data that describe the performance of different components/applications in HDCs. However, they entail significant computing overhead and require a long time to train and periodically retrain neural networks given the massive volume of traffic, the diverse types of application, and constant evolution in HDCs. Thus, their usage may be limited to only a subset of components or applications. Furthermore, time-series data may be subject to Simpson's paradox (Section 6.2), which can lead to a decrease in detection accuracy. For example, if only 1% of the traffic from an application experiences degraded performance, it may not be noticeable in time-series data. In addition, ranking anomalous performance metrics alone may not provide enough information for differentiating between different outages or issues in HDCs, as they may exhibit similar trends in certain performance metrics. For example, prolonged network latency could be caused by problems on a local host (e.g. insufficient write buffer), within the network (e.g. network congestion), or on a remote host (e.g. insufficient receive buffer). Therefore, further analysis is often necessary for identifying correlations between anomalous performance metrics or shared attributes among detected anomalous events.

2.2.2 Performance Statistics/Traffic Features Based Approaches

In addition to using time-series data, another approach for detecting performance anomalies in HDCs involves using distribution statistics and applying heuristic rules, statistical tools, and learning-based classification/clustering techniques. Further analysis is then conducted for characterizing the detected anomalies and identifying the source of the problem, such as links, switches, virtual machines, applications/microservices, or rollouts. Some recent studies have used this approach for anomaly detection in HDCs and have implemented strategies for anomaly characterization.

2.2.2.1 Heuristic Rules

Yu et al. (33) introduced a network application profiler, called *SNAP*, with low computation and storage overhead for classifying performance problems that occur in different stages of data delivery. It passively collects TCP statistics, including the total number of timeouts, the number of bytes in the send buffer, and the current congestion window, and the socket call logs, including the time and number of bytes whenever the socket makes a read/write call. With the above information, *SNAP* uses a set of heuristic rules derived by experts to identify four different performance problems: Sender application limited (the sender application may not generate the data fast enough, either by design or because of bottlenecks elsewhere), send buffer limited, network limited (including fast retransmission and timeouts), and receiver limited. *SNAP* can not only detect anomalous events, but it can also distinguish between different performance problems. After detecting individual connections that experience anomalous performance based on collected information, *SNAP* further characterizes them as resource constraints if they exist only among specific links/hosts/switches, or as application issues if they occur only within certain applications but on different machines and at different times. In order to do this, *SNAP* obtains topology and routing data to learn which connections share resources such as host, link, top-of-rack switch, or aggregator switch, and correlates performance problems across these connections. Specifically, it calculates the Pearson correlation coefficient (80) between performance problems of connections in the same application or sharing a host, link, or switch, aggregated over a correlation interval t , and reports correlated performance problems if the average correlation coefficient is above a predefined threshold. Through validations, *SNAP* correctly distinguished faulty servers and switches with injected problems from the performance of connections on other machines in a data center. After being deployed in a production data center with more than 8,000 servers and 700 applications (including MapReduce (81), storage, database and search services) over a week, *SNAP* finds 15 major performance problems in the application software, the server network stack, and the underlying network, greatly speeding up the debugging process.

2.2.2.2 Statistical Tools

Roy et al. (21) detected and located faulty links and switches in data centers based on full-path routing information and TCP metrics collected on end hosts, including the number of re-transmitted packets, congestion window (cwnd), slow start threshold (sssthresh), and smoothed round-trip time (srtt). They hypothesize that every observed value of a given metric under test can be treated as a sample of the same underlying distribution that characterizes a fault-free link. Given this hypothesis, during a particular time period with a given workload, faulty links will exhibit substantially different distributions from non-faulty ones. Therefore, determining a faulty link reduces to determining whether the samples collected on that link are part of the same distribution as fault-free links. The detection process contains four steps. First, the path that a packet traverses is identified and the statistics for the above TCP metrics along each link are calculated over successive fixed-length sampling periods. Second, samples of each TCP metric are assigned into per-link and per-direction buckets. Third, the distribution for each link is compared with the aggregate distribution for all other links for each metric with statistical analysis techniques (e.g., student's t-test (82) and KS test (83)). According to their hypothesis, non-faulty links are likely to have similar distributions in performance metrics with small variations. Hence, if the test distribution is sufficiently skewed to the right, the test link is considered as faulty and a verdict is issued to a centralized controller. Finally, the verdict results are analyzed and the faulty links are determined using the chi-square test (84) with the null hypothesis that, in the absence of faults, all links will have relatively similar numbers of hosts flagged as non-faulty. Hence, links/switches are considered as faulty if the number of guilty verdicts along specific links/switches is sufficiently large over a fixed accumulation period. The proposed technique is evaluated with induced packet losses and delay in a production Facebook front-end data center and a small private test bed. In both cases, it identifies faulty links/routers with high sensitivity and accuracy.

Although statistical analysis techniques have low overhead, obtaining full-path routing information for each flow can be complex, and the probing overhead increases exponentially as the number of hops and routing choices per hop increase. In addition, the threshold for issuing

verdicts may need to be adjusted to be more sensitive/conservative for detecting certain outliers. This may add more complexity as new outliers continue to arise with new computing paradigms and upgrades in data centers.

2.2.2.3 Learning-based Techniques

Gan et al. (35) developed *Seer*, an online cloud performance debugging system, to predict upcoming Quality of Service (QoS) violations as cloud services shift from complex monoliths to hundreds of loosely coupled microservices for meeting performance requirements and facilitating frequent application updates. Microservices is a software design approach that structures an application as a collection of small, independent services. Each service is responsible for a specific business capability and is developed, deployed, and scaled independently of the other services. *Seer* uses distributed RPC-tracing to monitor per-microservice latency and the number of requests queued in each microservice, and it learns spatial and temporal patterns from the large amount of tracing data generated by the microservices. To identify imminent QoS violations, *Seer* applies advanced neural networks, which are effective in pattern recognition and do not require prior knowledge of dependencies between individual microservices (85).

In *Seer*, each input and output neuron corresponds to a microservice, arranged in topological order. The back-end microservices are placed at the top, and the front-end microservices are placed at the bottom. The input describes the latency and number of queued packets for each microservice, and the output generates the probability of QoS violations. The neural network consists of a set of convolution layers, which are especially effective at reducing the dimensionality of large datasets and finding patterns in space, followed by a set of long-short-term memory (LSTM (44)) layers, which are particularly effective at finding patterns in time. Experimenting with five end-to-end services (social network, banking system, media services, hotel reservation site, and e-commerce service) and open-loop workload generators, *Seer* succeeds in foreseeing and avoiding 84 out of 89 QoS violations. Furthermore, *Seer* provides information on how to better design microservices for predictable performance by analyzing detailed statistics on

low-level hardware monitoring. It identifies problematic resources and detects recurring patterns that lead to violations of quality of service (QoS). Specifically, *Seer* examines CPU, memory capacity and bandwidth, network bandwidth, cache contention, and storage I/O bandwidth (35) to identify saturated resources. When this information is inaccessible, *Seer* instead uses a set of 10 tunable contentious microbenchmarks, each of them targeting a different shared resource (86), to determine resource saturation. Using this method, application developers can not only anticipate upcoming QoS violations, but also better understand bugs and design features of microservices that lead to hotspots. Examples of these hotspots include microservices with a lot of back-and-forth communication, microservices that form cyclic dependencies, and microservices that use blocking primitives. However, the training process may take several hours up to a day for week-long traces collected on a 20-server cluster, which may greatly limit its application in production HDCs, which host thousands of servers and hundreds of applications. Additionally, constant retraining is required, especially when there are major changes to the microservices.

Arzani et al. (40) introduced *NetPoirot*, which uses learning-based techniques to identify the responsible entity (server, client, or network) for failures at individual endpoints in data centers, rather than relying on hand-crafted heuristic rules. The network is often unfairly blamed when performance failures occur in data centers, leading to long and expensive debugging processes that involve many engineers and developers in different organizations. By revealing the most likely source of the failure within the infrastructure, the team (client, network, or remote service) responsible for the failure can provide a timely response, rather than having the task of troubleshooting pass around within organizations. To achieve this goal, *NetPoirot* monitors an extensive set of raw TCP statistics from each socket (e.g., number of flows, maximum congestion window, number of bytes sent by TCP). These statistics are aggregated based on destination IP/port in TCP connections, which are then used as input to the supervised learning algorithm known as random forest (87). In order to construct representative training samples for learning purposes, known failures are injected into endpoints from time to time and the corresponding TCP statistics are captured into endpoints together with the faulty entity as ground truth. To im-

prove performance, *NetPoirot* uses the random forest algorithm and multiround classification to first identify the failed *network*, then the failed *server*, and finally the failed *client*. The effectiveness of *NetPoirot* is evaluated using two test applications (a duplex application and a simplex application with diverse communication patterns) and two real-world applications. It shows promising performance with high precision and recall. Compared to neural networks, the computing overhead and training time of random forests are smaller.

However, each application with a different communication pattern and reacting mechanism must maintain a separate model that is trained with injected faults to ensure its performance. Additionally, supervised learning approaches require a representative labeled dataset, which can be costly to create and may quickly become outdated. This is because these approaches are limited to prior knowledge and cannot detect unseen events in evolving HDCs.

2.2.2.4 Rollout Impact Analysis

Modern data centers undergo constant changes as systems evolve. However, these changes can unexpectedly degrade the performance of applications (88; 31). For example, Google reports that 68% of the failures occur during network changes (88). To prevent widespread performance outages during upgrades in cloud-scale infrastructures such as Azure (3), Li et al. (31) introduced *Gandalf*, a tool that assesses the impact of software roll-outs on application performance. *Gandalf* follows three steps: (1) Obtain faulty signals from raw telemetry data (e.g., OS events, log messages, and API call statuses) and detect anomalies based on the occurrences of each fault signature. Ambient faults, such as hardware and network glitches, are common in a large-scale cloud system. *Gandalf* estimates baseline fault occurrences from past data using Holt-Winters prediction (89) and marks samples that deviate from the expected value by more than 4 standard deviations as anomalies. (2) Correlate each fault signal with ongoing rollouts spatially and temporally to determine which rollout may have caused the fault signals. Rollout are more likely to be responsible for a fault if they occur approximately together on the same nodes in the system. (3) Train a Gaussian discriminant classifier (90) to assess the impact of the fault. The results decide

whether the rollout should be stopped or continued, based on the training data generated from historical deployment cases with feedback from component teams. After being deployed in Microsoft Azure for more than 18 months, *Gandalf* achieves 92.4% precision and 100% recall for data-plane rollouts, and 94.9% precision and 99.8% recall for control-plane rollouts, effectively blocking bad rollouts and preventing catastrophic service outage and customer impact. However, after faulty rollouts are paused, more analysis is still required to understand why they may have caused abnormal behavior and how to address it.

2.2.3 Dependency-graph Modeling

Research in this area explicitly models the dependency/connectivity between components in HDCs using dependency graphs, and then applies probability propagation or inference analysis to identify the root-cause components that explain most of the anomalies.

Paramvir et al. (91) introduced an Inference Graph model (92) to represent dependencies of packet traces and designed *Sherlock* to create Inference Graphs and localize performance problems in large enterprise networks. The Inference Graph model is a labeled, directed graph that provides a unified view of the dependencies in an enterprise network, spanning services and network components, and is based on packet traces collected at agents/routers. Components in the network are classified into three types: root-cause nodes that correspond to physical components and can cause end-users to experience failures, observation nodes that model a user's experience when using services in the network and can be measured by *Sherlock*, and meta-data nodes that act as glue between root-cause nodes and observation nodes. Each node has three potential states: *up* (services work normally), *down* (services that experience a fail-stop failure), and *troubled* (a subset of services experience failure). To explain how the state of a parent node influences the state of a child node, *Sherlock* introduces three types of meta-nodes: noisy-max, selector, and failover. These meta-nodes model the dependencies between root causes and observations. For noisy-max, if any of the parents are in the *down* state, then the child is also *down*. The selector and failover meta-nodes are used to model load balancers and failover redundancy, respectively.

Sherlock also introduces an algorithm called Ferret to identify root-cause nodes. Ferret measures the agreement between the measurements collected from the observation nodes and the propagated probability calculated by exploring different state assignments to each root-cause node of being *up*, *down*, or *troubled*. For example, Ferret can specify that *link*₁ is *troubled*, *server*₂ is *down*, and all other root cause nodes are *up*. Taking the Inference Graph and the measurements (e.g., response times) associated with the observation nodes as input, Ferret generates a ranked list of assignment vectors ordered by a confidence value that represents how well they explain the observations. During a 5-day period in an organization’s production network, *Sherlock* identifies more than 1,029 network performance problems. Moreover, it effectively traces more than 87% out of 350 potential blames to just 16 root causes.

Based on (91), Kandula et al. (93) introduced multiple variables to describe each component of the network, rather than a single abstract variable. These multiple variables capture different aspects of behavior in each component, including resource consumption, response time for queries, and application-specific aspects such as the fraction of responses with error codes. In addition, they allow components to interact with each other in complex ways, depending on their state, in order to observe and diagnose a rich set of failure modes. With these improvements, Kandula et al. (93) build *NetMedic*, which uses the joint behavior of two components in the past to estimate the likelihood of them impacting each other in the present. Specifically, *NetMedic* searches the history of component states for time periods in which the state of the source component is “similar to” to its current state. If during those periods the destination component is often in a state similar to its current state, it is likely that the source component in the current state is impacting the destination component. Hence, given a component whose visible state has changed relative to some period in the past, *NetMedic* is able to identify the components that are likely responsible for the change. The deployed prototype effectively diagnosed faults through a chain of dependency edges injected into a live environment with roughly 1,000 components and 3,600 edges, with each component populated by roughly 35 state variables. The faulty component is

correctly identified as the most likely culprit in 80% of the cases and is almost always on the list of the top five culprits.

The performance of previous fault location algorithms varies significantly in terms of run time and accuracy for different networks, especially for those with distinct characteristics. Unfortunately, there is no algorithm that provides both high localization accuracy and low computational overhead. Mysore et al. (34) devised *Gestalt*, a new algorithm that combines the best features of existing fault localization algorithms and includes a new technique for exploring fault hypotheses. *Gestalt* is capable of achieving good performance in many networks and conditions with reduced computational cost by navigating a continuum between the extremes of greedy failure hypothesis exploration (94) and combinatorial exploration (91) to explore the space of fault hypotheses. By running experiments on three real, diverse networks, *Gestalt* achieves either a significantly higher localization accuracy or an order of magnitude lower running time compared to three existing fault localization algorithms.

Zhang et al. (9) designed *Deepview* for virtual hard disk (VHD) failure localization. VHD failure is caused by the separation of virtual machines (VMs) and their VHDs and has become a frequent failure that severely reduces VM availability. For example, among thousands of VM down events every day in Microsoft Azure, 52% of them are caused by VHD failures. *Deepview* gathers VHD failure events, VHD paths between VMs and their storage as input and constructs a bipartite model to connect the compute, storage, and network components together. With the bipartite model, an algorithm that integrates Lasso regression (95) is designed to select a small subset of components as faulty candidates and apply hypothesis testing (96) for deciding which component among compute, storage, and network is most likely to be responsible for VHD failures. Specifically, if a component's Lasso estimate is much worse than the average, it is likely to be a real failure and is flagged. Once the location of the failure is confirmed, the responsible team often has standard procedures for quick mitigation. After being deployed on Microsoft Azure, *Deepview* succeeds in reducing the number of unclassified VHD failure events from tens of thousands to several hundreds, and the time-to-detection for incidents to less than 10 min-

utes. In addition, *Deepview* reveals new patterns related to VHD failures, including unplanned top-of-rack switch (ToR) reboots and storage gray failures.

Although dependency graph-based modeling approaches are effective in localizing faulty components in data centers, they require explicit modeling of dependencies/connectivity, making these less scalable in large, adaptive, and evolving HDCs (35).

2.2.4 Limitations

These anomaly detection and characterization systems/algorithms can be quite successful in detecting abnormal performance based on historical data and aggregating large sets of anomalies into groups emanating from the same application/material/link, helping to reduce the scope of diagnostics for network engineers. However, the analysis techniques used are either black-box learning approaches, which are inherently uninterpretable, or require explicit modeling of dependencies/connectivity, which is not scalable in large, adaptive, and evolving HDCs. In an HDC environment, consequently, it may still take days or even weeks to determine the root cause of performance problems or the potential impact of a planned infrastructure change.

Moreover, current systems do not provide insight into normal performance data, which is necessary for understanding the network performance of HDC applications and infrastructure components for assessment and planning purposes. For example, what are the ratios of traffic from an application constrained by compute, storage, or network resources? What is the average latency incurred for applications on end hosts and over the network? Answering these questions allows us to create a performance profile for HDC applications that can be used not only for anomaly detection and characterization, but also for application design, job allocation, and network upgrades. For example, bandwidth-constrained and computation-constrained applications may be placed in the same cluster to maximize HDC resource utilization and reduce resource contention. If the performance of most applications is constrained by computing resources, more CPU resources can be deployed to alleviate this constraint. Rather than relying on black-box tech-

niques, such analysis requires dimension reduction approaches that retain the physical semantics and interpretability of performance data.

Our goal is to develop interpretable models for understanding the network performance of applications in HDCs. These models can aid in the assessment, planning, and diagnosis tasks in HDCs. In next chapters, we will discuss our approach (Chapter 3) and implementation (Chapter 4) in detail.

CHAPTER 3: OUR APPROACH: CONSTRAINT-BASED MODELS BASED ON REMOTE PROCEDURE CALL TELEMETRY

In this chapter, we describe our approach for creating a modeling tool to understand the network performance of applications in HDCs. Our tool is designed to handle the enormous scale, tremendous diversity, and complex couplings present in HDCs, and to assist assessment, planning, and diagnosis tasks.

The structure of this chapter is as follows. In Section 3.1, we motivate the use of a constraint-based modeling approach for analyzing the network performance of HDC applications. In Section 3.2, we describe the performance data that should be collected for identifying potential constraints experienced by these applications. In Section 3.3, we present the modeling technique we have chosen for modeling performance constraints.

3.1 The Goal: Constraint-Based Modeling

Analyzing and understanding the network performance of applications in HDCs is a major challenge due to the massive scale of the performance data involved. Firstly, this data represents billions of network transfers from thousands of applications running in different parts of an HDC. Even when focusing on only a few applications or only a small portion of the HDC infrastructure, engineers may need to consider performance data for tens of millions of daily network transfers, which can be overwhelming. Secondly, multiple performance metrics are associated with each network transfer, such as latencies, rates, volumes, and losses. The statistical derivatives of these metrics alone can easily number in the hundreds. These metrics are measured at different vantage points, such as senders, receivers, and network subsystems, further increasing the scale of the analysis. Thirdly, HDCs host thousands of applications that provide different types of services,

have diverse workloads and service level objectives (SLOs), and may run on differently configured parts of the HDC infrastructure. Each of these aspects may impact performance analysis. For a comprehensive understanding of network performance, it is essential to collect performance data from a wide range of applications running on diverse infrastructures. This further increases the scale of the data that needs to be analyzed.

Analyzing the performance of billions of distributed workers in HDCs is a challenging task because the performance of these workers is influenced by numerous factors. It is impossible to gain a comprehensive understanding of the network performance of an application by analyzing the performance of a single application worker alone. Therefore, we focus on understanding the *distribution* of performance across different dimensions, using statistical measures such as the mean to represent average performance and the spread to indicate variability. By analyzing performance distributions aggregated over a larger number of network transfers, rather than examining each transfer individually, we can also effectively and succinctly summarize the performance of thousands of workers.

Despite the significant reduction in the number of network transfers required for analysis, there are still two challenges that must be addressed when analyzing performance distributions. The first challenge is selecting an appropriate aggregation approach for grouping together network transfers to study using distributions. The second challenge is the complexity of having to analyze the distributions of hundreds of performance metrics. To overcome these challenges, we propose characterizing network performance of applications by focusing on the bottleneck constraints that limit their performance. As explained below, this approach allows us to group network transfers experiencing similar network performance, reduce the number of performance metrics that need to be analyzed individually (dimension reduction), and make our analysis more interpretable (interpretability), which are crucial for addressing the scale, diversity, and complexity present in HDCs.

Dimension Reduction In HDCs, there are billions of distributed workers, each with hundreds of performance metrics. However, only a small number of possible bottleneck constraint types

can limit the network performance of application workers. For example, the bottleneck could be limited processing or transmission capacity at the sender, receiver, or network switches; or it could be sluggish mechanisms in the transport protocol; or it could be limitations in application's data-generation behavior itself. Groups of application workers engaged in similar computations in similar parts of the infrastructure are likely to experience similar bottleneck constraints. By characterizing the performance of HDC application workers based on their bottleneck constraints instead of individual performance metrics, we can simultaneously characterize the representative performance of a large number of workers. This offers intrinsic and significant dimension reduction, and allows us to “see the forest” rather than focus on individual leaves and trees, as we demonstrate later.

It is important to note that the bottleneck constraints determine how different performance metrics are coupled and how these metrics are simultaneously impacted by certain factors. Hence, analyzing constraints boils down to *simultaneously* analyzing multiple performance metrics, which is important for understanding the network performance of applications.

Interpretability Interpretability allows engineers to glean meaningful insights from performance data, which plays a crucial role in facilitating assessment, planning, and diagnosis tasks. Since HDC applications are designed to run *as fast as possible* by fully utilizing available resources (49; 50), the performance of their application workers is always determined by constraints that bound what is “possible”. Understanding these bottleneck constraints can lead to useful insights and inform decision-making. For example, determining whether the performance of most applications in one part of a data center is limited by bandwidth resources can suggest whether the fabric bandwidth should be increased to improve performance or reduced to save cost (assessment and planning). Additionally, understanding how the bottleneck constraints change after a planned upgrade or during an unexpected performance outage can help identify the cause or solution for anomalies (diagnosis), since resolving a performance issue often involves alleviating a bottleneck constraint.

In summary, our goal is to use a constraint-based modeling approach to characterize the network performance of HDC applications in a way that achieves both dimensional reduction and interpretability. In the following sections, we will discuss the data collection and modeling techniques that allow us to achieve this goal.

3.2 Performance Data: Remote Procedure Call Telemetry

In production HDCs, several high-level application performance metrics are routinely collected, such as queries per second (QPS), delivery rate, and response times. While these metrics are efficient for detecting anomalies (19; 21), they often only provide partial information, which hinders thorough investigations and better interpretability. For example, a significant reduction in QPS could be caused by several factors, such as exhausted resources on end hosts, insufficient bandwidth along routing paths, misconfigured transport protocols, or even bugs in applications. Each of these issues must be addressed differently for resolving the problem. Other network telemetry, such as link utilization or queueing latency, provides information about the aggregate network load but does not distinguish the performance of one application from another. To improve interpretability at the application level, it is necessary to use lower-level performance metrics that are related to HDC application workers and can reflect their bottleneck constraints.

3.2.1 How Constraints Manifest in HDCs

In HDCs, there are three main subsystems that can constrain the network performance of distributed application workers: the local host, which can be limited by CPU, memory, and application data generation; the remote peers and resources with which they communicate, which can be limited by CPU, memory, and application data consumption; and the network infrastructure, which can be limited by link capacities, buffer sizes, processing powers, and switch fabric architectures. Constraints manifest themselves mainly in three types of performance metrics observed within these subsystems: (1) *Rate*, which is influenced by link capacities, cross-traffic, and processing speeds. Examples of rate metrics include delivery rate (throughout of a connection)

and pacing rate (the speed at which congestion control algorithms send packets). (2) *Latency*, which includes queueing delay, serialization delay, propagation delay, and processing latency. The queueing delay is upper bounded by the maximum buffer size on switches and routers along network paths and affected by the volume of cross-traffic. The serialization delay depends on traffic volumes and NIC speeds. The propagation delay is determined by the physical propagation distance between the local and remote host. The processing latency is primarily influenced by the availability of CPU and memory resources. (3) *Volume*, which describes the amount of data transferred by the sender. Examples of volume metrics include RPC sizes, congestion window sizes, and bytes-in-flights. Volume metrics are largely determined by the data generation behavior of applications and the underlying network transport protocols, such as Transmission Control Protocol (TCP) (97) and User Datagram Protocol (UDP) (98). Physical constraints in HDCs impose hard limits on one or more of these performance metrics. For example, the minimum delay a network transfer can achieve is limited by the propagation delay, as nothing can travel faster than the speed of light. Similarly, the maximum delivery rate one can obtain is limited by the bottleneck NIC speed.

Because performance metrics are often coupled, each of the metric types listed above only provides partial information about constraints. For fully understanding bottleneck constraints, it is necessary to consider these types of metrics simultaneously. For instance, the delivery rate can be improved by increasing the sending rate and congestion window without affecting end-to-end latency, but only up to a limit (the bandwidth-delay product). Beyond this limit, the delivery rate remains constant and latency increases. In the first case, the constraint is the sending application worker (the application is not generating enough data to fully utilize the available bandwidth along the network path). In the second case, the constraint is the network (the available bandwidth is fully utilized and queues form along the network path) (99). Therefore, for revealing constraints, we need to measure rate, latency, and volume metrics in these three subsystems simultaneously.

3.2.2 A Vantage Point for Observing Constraints: Remote Procedure Call

Remote Procedure Call (RPC) is a protocol that enables efficient communication between services within and across data centers. It allows an application program to call a procedure regardless of whether it is local or remote. RPC uses the request/reply paradigm, in which a client sends a request message to a server and the server responds with a reply message, with the client suspending execution while waiting for the reply. RPC is a commonly used mechanism for building distributed systems and is a fundamental component of many applications in HDCs (12).

A good vantage point for observing performance metrics related to rate, latency, and volume in all three subsystems (the local host, the remote host, and the network) is applications' RPC library. For example, metrics such as the RPC request inter-arrival time and sender queueing latency (such as latency in the TCP buffer and queueing discipline layer) can be used to quantify local host constraints. The response service time can be used to quantify remote constraints, and metrics such as transport RTT and delivery rate can be used to quantify network constraints. By recording these performance metrics through instrumentation of the network stack on a per-RPC basis using the application's RPC library, we can gain a comprehensive view of applications' network performance and determine which subsystem is the most constrained and why, as explained below.

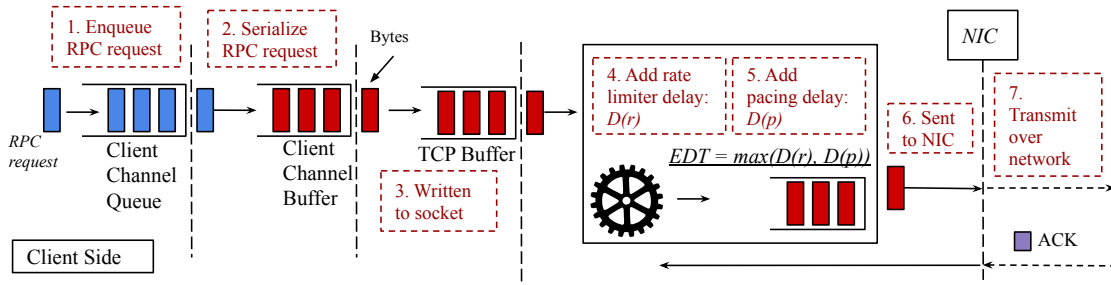


Figure 3.1: RPC workflow: Client Side

3.2.2.1 The RPC Workflow

Figure 3.1 illustrates the procedure for sending an RPC request from a local host in one of the largest production HDCs. This process generally consists of six steps on the local host:

1. Enqueue the RPC request into the client channel queue;
2. Serialize the RPC request into bytes and store it in the client channel buffer;
3. Write the serialized bytes in the TCP buffer;
4. Add rate limiter latency $D(r)$ to enforce different transmission priorities (Differentiated Services Code Point (100) priority) for each user (101);
5. Add pacing latency $D(p)$ to smooth out the traffic and avoid network congestion (99);
6. Send the bytes to the NIC based on its earliest departure time (EDT), which may depend on the maximum values of $D(r)$ and $D(p)$.

The bytes then leave the local host and traverse the network. Upon reaching the server side, the received bytes are stored in the TCP recv buffer, and an acknowledgment (ACK) is sent back to the client to notify the successful receipt of the data. The RPC request is then enqueued into the server buffer before being deserialized and processed further (Figure 3.2). After the server generates the responses, it repeats the steps 1-6 as shown in Figure 3.1 to send responses back to the client over the network.

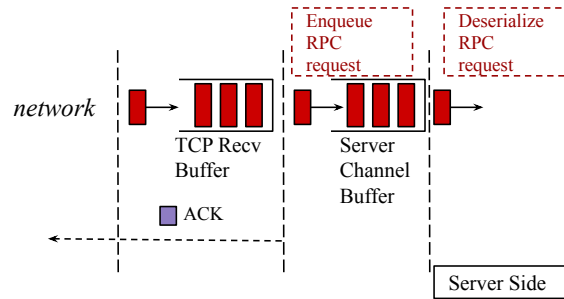


Figure 3.2: RPC workflow: Server Side

3.2.2.2 Potential Constraints for an RPC

Neal et al. (99) study the network subsystem and identify three potential constraints that could affect the network performance of applications: the data generation rate of the application, the bottleneck network bandwidth, and the buffer space at the bottleneck link. If the data generation rate is low, the application network performance is constrained by its own data generation rate, as there is not enough data to transmit. However, if the data generation rate is high, the application network performance becomes constrained by the bottleneck network bandwidth, leading to a queue at the bottleneck and a linear dependence of Round-Trip Time (RTT) on the amount of data in flight (data that has been sent but not acknowledged). As the queue length continues to increase and exceeds the buffer capacity, packets start to drop and performance becomes constrained by available buffer space. These constraints can be identified by simultaneously analyzing trends in delivery rate, RTT, amount of data in flight, and packet loss rate.

However, further examination of the stages involved in sending and receiving RPCs on hosts (illustrated in Figures 3.1 and 3.2) reveals additional potential bottlenecks that can limit the network performance of HDC applications:

- **Local CPU/memory-limited constraint.** The processing delay for enqueueing and serializing RPC requests depends on the availability of CPU/memory resources on local machines. Therefore, this delay can increase if these resources are insufficient on the client side (stages 1 and 2 in Figure 3.1);
- **Rate limiter-limited constraint.** The rate limiter enforces different transmission priorities among applications on a shared network by maintaining multiple queues with different priorities. Traffic in different queues is served in a round-robin fashion, each with a different allocated ratio. For instance, in each round, it may serve four packets from a high-priority queue, but only one packet from a low priority queue. Therefore, if an application has a low transmission priority when competing for shared bandwidth or is overwhelmed by volumes

of traffic from other applications with the same priority, it may experience high rate limiter latency (stage 4 in Figure 3.1) ;

- TCP congestion control algorithm-limited constraint. There are two main transport protocols, TCP and UDP, in data center networks. Most data center traffic is TCP-based (102). TCP congestion control algorithms are used by each sender for determining how much capacity is available in the network and how many packets can be safely in transit without causing congestion (stage 5 in Figure 3.1). Therefore, the network performance of applications can be limited on end hosts if the TCP congestion control algorithm is not well-suited to the HDC environment and traffic patterns. For example, a too conservative TCP congestion control algorithm may not send data fast enough to fully utilize available bandwidth resources.

New TCP congestion control algorithms have been developed to better handle the evolving traffic patterns in data centers, such as CUBIC (103), Data Center TCP (DCTCP) (104), and BBR (99). In this work, we focus on BBR (99), a recent algorithm that has been shown to achieve high performance with HDC traffic and has been adopted by Google, one of the largest production HDCs in the world. BBR controls two main parameters for avoiding congestion: the congestion window size, which determines how much data can be in transit at a given time, and the pacing rate, which decides how fast data can be sent out. Therefore, the constraint can be further divided into a limited congestion window constraint (stage 3 in Figure 3.1), or a pacer-limiting constraint (stage 5 in Figure 3.1);

- Remote CPU/memory-limited constraint. Similar to the client side, the server-side processing delay may increase if the workload of enqueueing and deserializing RPC requests exceeds the capacity of the available CPU/memory resources (Figure 3.2).

Given the RPC workflow in Figures 3.1 and 3.2, there are a total of seven types of constraints that an RPC can encounter in these three subsystems. Each constraint is associated with at least one of the rate, latency, and volume metrics, as shown in Table 3.1. Measuring these metrics in

the RPC library can help identify constraints and provide insight into the network performance of HDC applications.

Constraint Type	Performance Metrics
Application-limited	delivery rate, RTT
Network bandwidth-limited	delivery rate, RTT
Buffer capacity-limited	packet loss rate
Local CPU/memory-limited	sender queueing latency
Rate limiter-limited	rate limiter latency
TCP congestion control algorithm-limited	pacing latency, congestion control window size
Remote CPU/memory-limited	receiver queueing latency

Table 3.1: Constraint types and their associated performance metrics in Figures 3.1 and 3.2.

Constraint Types Are Application-Agnostic Given a fixed RPC workflow, the set of potential constraint types that can limit the network performance of RPCs are consistent across different applications. Therefore, when analyzing performance constraints from RPC telemetry data, it is not necessary to consider the specific applications that generate RPCs or the dynamics among different applications. Furthermore, once the constraint type has been identified, application-level information such as workload, service types, transmission priority, and locations, can be used to understand the cause of the constraint. For example, a low transmission priority in the network can explain why the network performance of RPCs is constrained by the higher rate limiter latency (stage 4 in Figure 3.1).

To summarize, by collecting telemetry data from RPC, we can gain a comprehensive understanding of the network performance of applications in HDCs, as most applications rely on RPCs to fulfill their services. However, analyzing hundreds of rate, latency, and volume metrics for billions of RPCs from thousands of applications in HDCs can be a challenging task due to the sheer scale of the data involved. In order to effectively address this challenge, modeling techniques that can reduce the dimensionality of performance data while still preserving crucial information for interpreting performance constraints are needed. In the following section, we introduce our modeling approach for identifying constraints based on per-RPC performance metrics. This approach provides both dimension reduction and interpretability.

3.3 Interpretable Modeling: Gaussian Mixture Models (GMMs)

3.3.1 Background: Gaussian Mixture Models

Gaussian mixture models (GMMs) (48) are powerful probabilistic models used for describing the distribution of data, assuming that the data is generated from a mixture of a finite number of Gaussian distributions. Each of these Gaussian distributions, known as a “blob”, represents a subset of data that exhibits distinct characteristics. GMMs are useful for identifying patterns and structures in data, and have a wide range of applications in fields such as machine learning, data mining, and pattern recognition. In our case, we use GMMs to analyze RPC telemetry data in HDCs. Each blob in the GMM corresponds to a subset of RPCs with similar performance characteristics, which are different from that of RPCs in other blobs. The number of blobs in a GMM is typically predefined. Each blob is characterized by three parameters: mean, covariance, and mixing probability. The means and covariances describe the location (center), shape (width), and orientation (correlation between multiple dimensions) of the blob, while the mixing probability indicates the contribution of each blob to the overall mixture model. The parameters of the GMM can be estimated using the expectation-maximization (EM) algorithm (105). By analyzing the GMM, we can understand patterns and structures in the RPC telemetry data and gain insight into the network performance of HDC applications.

3.3.2 Interpretable Modeling

Our approach is inspired by (106), which argues that to achieve interpretability, a model should “...obey structural knowledge of the domain, such as monotonicity, causality, structural constraints, additivity, or physical constraints that come from domain knowledge.” Therefore, the key question is *what is the appropriate physical model for the network performance of aggregates of distributed HDC workers?*

The performance measurements of the rate, latency, and volume of each RPC are each limited by different physical constraints. For example, the delivery rate can only increase up to the

bottleneck link capacity, the end-to-end latency can never be smaller than the propagation delay based on the physical distance between end hosts and the speed of light, and the traffic volume can never exceed what applications generate. If we consider the multi-dimensional space defined by these metrics, then coexisting physical constraints can be represented as a high-dimensional polytope. Each vertex on the polytope sets a limit beyond which at least one physical constraint is violated and is unlikely to occur in reality. Therefore, distributed application workers are only able to operate within specific regions in the multidimensional constrained space, depending on their service types and implementations, as well as the underlying hardware, software, and network configurations in HDCs.

Different constraints interact because the usage of HDC resources is coupled as discussed in Section 1.1.2. For instance, a relief in the application-limited constraint (an application is not generating enough data to fill the pipe along the routing path) may lead to the bandwidth-limited constraint and increase the end-to-end latency, while an increase in bandwidth may put more pressure on the remote end and result in the remote CPU/mem-limited constraint. Given that distributed application workers are usually designed to run *as fast as possible* to maximize resource utilization, their network performance will be constrained by one or more factors in local, network, and remote systems. Optimizing performance under these different and interacting constraints is essentially a linear programming problem. The steady-state solutions are located at the vertices of the multidimensional polytope described above, which is defined by constraints. That is, the performance of workers eventually converges to a steady-state point, as they fully exploit the bottlenecked resources.

HDC applications scale their computations by spawning a large number of distributed workers. Therefore, there will be a large number of application workers that perform similar processing (in different data slices), share a common transmission priority, and run in similar parts of the HDC infrastructure (10; 11). Such a group of workers is likely to be constrained in the same way. As a result, application workers experiencing similar constraints will operate near the same

vertex in the multidimensional polytope in an independent¹ and identically distributed (IID) manner. Based on the central limit theorem, the sum of multivariate IID distributions is multivariate normal (Gaussian) (108) — with the mean representing the average behavior of these workers and the spread describing the degree of variations they may experience due to noise and temporal dynamics in HDCs. Workers in different applications (e.g., monitoring service or streaming service) or performing different operations within the same application (e.g., MapReduce (81; 109) or Spark (109)) may be constrained in different ways and will operate near different vertices in the multidimensional polytope. Because high-dimensional polytopes are spiky, the vertices are expected to be well separated (110). Therefore, a multidimensional Gaussian mixture model (GMM) (48) is an appropriate physical model for characterizing the underlying distributions of workers constrained by different bottlenecks in the multidimensional metric space.

Based on the above discussion, we propose using GMMs to model the network performance of HDC applications, by utilizing per-RPC performance metrics about rate, latency, and volume. Such a GMM-based modeling approach offers three key advantages:

1. Tremendous dimension reduction. GMMs automatically distill measurements of thousands of HDC application workers, each with hundreds of performance statistics, into a handful of multivariate Gaussians (referred to as *blobs*). Each blob represents a unique network performance experienced by applications under different constraints. This provides a concise summary and greatly reduces analysis overhead.
2. Application- and topology-agnostic. GMMs can separate application workers experiencing different network performance into different blobs, without requiring prior knowledge or assumptions about the applications or the underlying HDC infrastructure. This greatly reduces the manual effort needed to determine the appropriate level of aggregation for

¹For achieving optimal performance and avoiding correlated failures, HDCs are designed to minimize interference between application workers. For example, the use of a multistage non-blocking Clos topology provides substantial in-built path diversity and redundancy (28). Duplicated hardware, such as power supplies, servers, and cooling systems, are leveraged in HDCs to support jobs running in every part of the data center. Scheduling policies that spread tasks of a job across failure domains, such as machines, racks, and power domains, are employed to further increase resilience (107). These efforts help minimize spatial correlations among application workers, leading to independent performance for each.

the large number of performance metrics from billions of RPCs, and eliminates potential negative impacts of Simpson’s paradox (38) introduced by human interventions.

3. Interpretability. GMMs preserve the physical semantics of each performance metric and help to interpret bottleneck constraints through the joint distributions of metrics within each blob. In contrast, previous studies that applied *black-box* approaches, such as advanced neural networks (45; 46; 47), lost the semantics and physical interpretability of the original performance metrics when the original high-dimensional feature space was projected onto a low-dimensional latent space.

In summary, our approach involves utilizing a multidimensional Gaussian mixture model (GMM) based on RPC telemetry data for modeling the bottleneck constraints that limit the network performance of HDC applications. This approach effectively reduces the dimensionality of the data, increases stability, and improves interpretability, making it suitable for handling the large scale, diversity, and complex couplings often present in HDCs.

Our work consists of four main stages: (1) Implementing a modeling pipeline that takes billions of RPC telemetry entries as input and generates a GMM for describing different network performance experienced by these RPCs. (2) Interpreting the modeling results through statistical tools. (3) Validating the effectiveness of our approach on a controlled test bed. (4) Implementing our approach in production HDCs and evaluating its efficacy in facilitating assessment, planning, and troubleshooting tasks. The subsequent chapters provide more detailed information on each of these stages.

CHAPTER 4: THE MODELING PIPELINE

In this chapter, we introduce the pipeline for modeling and analyzing the network performance of HDC applications based on bottleneck constraints. The pipeline takes RPC telemetry data as input and consists of two main components: GMM modeling and GMM analysis, as shown in Figure 4.1.

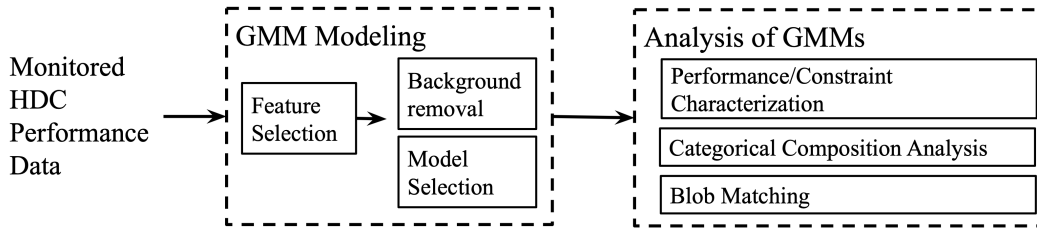


Figure 4.1: Modeling Pipeline

The GMM modeling component takes RPC telemetry data as input and generates a GMM that characterizes different observed behaviors in RPC network performance. Its focus is on producing accurate modeling results from the input data. The component includes three main elements: feature selection, model selection, and background removal. Feature selection determines which performance metrics are used for building GMMs. Model selection chooses the number of blobs in the GMM. Background removal eliminates any noises/outliers from the performance data. The GMM analysis component employs statistical techniques to analyze the results of GMM modeling. This analysis aims to characterize the performance associated with each blob and to identify their constraints. The analysis component is composed of three elements: performance characterization, categorical composition analysis, and blob matching. Performance characterization identifies key performance metrics and constraints of each blob in a GMM. Categorical composition analysis analyzes shared attributes of RPCs in each blob for understanding which services and jobs are constrained and what factors may be causing these impacts. Blob

matching ensures an “apples-to-apples” comparison between multiple GMMs for evaluating the impacts of infrastructure or upgrades on the network performance of HDC applications.

4.1 GMM Modeling

The GMM modeling component takes RPC telemetry data as input and produces accurate modeling results through three main tasks: feature selection, model selection, and background removal. These three tasks are crucial for producing accurate modeling results, which are used to gain insight into the network performance of RPCs. In the following section, we delve into each task in more detail.

4.1.1 Feature Selection

The total number of performance metrics gathered from RPC telemetry can be quite large, potentially numbering in the hundreds. These metrics include the distribution of different performance metrics such as rate, latency, and volume, captured across a wide range of percentiles, from the 1st to the 99th percentile, for an aggregated interval. Additionally, the telemetry system records several scalar metrics, such as the number of lost packets, the size of the RPCs, and the total number of RPCs.

Using a large number of performance metrics for building a GMM can lead to increased computational complexity, as the cost of modeling increases exponentially with the number of features. This phenomenon is known as the curse of dimensionality (111). For reducing computational expense, it is necessary to limit the number of performance metrics used in the modeling process. Techniques such as PCA (112), t-SNE (113) and UMAP (114) can be used to efficiently reduce the dimensionality of higher-dimensional data points by projecting them onto a lower-dimensional space. However, these techniques can result in the loss of interpretable semantics of the original dimensions after projection. Other techniques that rely on validation data with ground-truth information about how many blobs are needed and which measurements should be grouped in same/different blobs, such as those proposed in (115), may not be applicable in

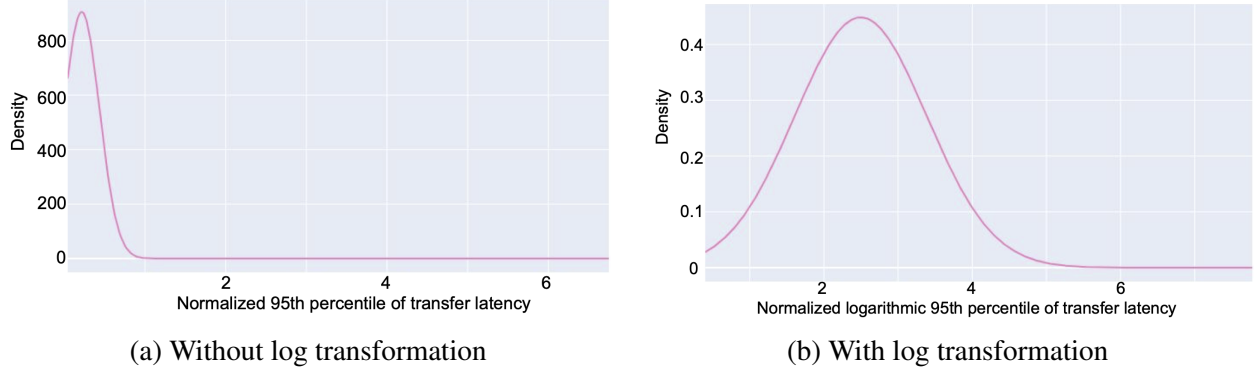


Figure 4.2: Probability density plot of transfer latency in one data center. Values on the x-axis are hidden due to proprietary reasons.

this research due to the absence of such information. In our research, we select the performance metrics for modeling by relying on both domain knowledge to identify metrics that are relevant to network performance constraints, and statistical analysis to assess them for normality using statistical tests (116). This approach allows us to select a set of performance metrics that accurately characterize the network performance of RPCs and can be effectively modeled using GMMs with manageable computational complexity.

GMMs assume that the data being modeled is generated from a mixture of several Gaussian distributions, which may not always hold true for metrics collected in RPC telemetry. For example, the distribution of RPC sizes may be influenced by the characteristics of the applications, such as their type of service, workload, and implementation, rather than conforming to a mixture of Gaussian distributions. Including non-Gaussian performance metrics for building GMMs may decrease the effectiveness of accurately identifying patterns and structures across multiple performance dimensions. For identifying and excluding such metrics, we use quantile-quantile (Q-Q) plots to assess the normality of the metrics. We consider the following three distributions as appropriate for modeling with GMMs: normal, multimodal, and log-normal.

Performance constraints in HDC applications are typically reflected in three types of performance metrics: rate, volume, and latency. Therefore, these metrics should be included when modeling GMMs for identifying constraints. By analyzing performance metrics collected in a production HDC, we find that volume and rate-related metrics often conform to normal or multi-

modal distributions, while latency-related metrics often follow log-normal distributions with long tails, due to the additive nature of latency. Specifically, if packet A is queued after packet B, A inherits the latency experienced by B due to packets queued ahead of B, in addition to the latency introduced by B. Therefore, we apply a logarithmic transformation to latency-related metrics to make their distributions conform to normality, making them more suitable for modeling with GMMs. Figure 4.2 shows probability density plots of transfer latency for an application in the production HDC. The logarithmic transformation leads to a better conformity to normality in the distribution, as seen in Figure 4.2(b), compared to the distribution in Figure 4.2(a).

For reducing the dimensionality of the input performance data, we use the Pearson correlation coefficient (80) (Eq. 4.1) to group highly correlated performance metrics and select a representative metric from each group (117).

$$r = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum (x_i - \bar{x})^2 \sum (y_i - \bar{y})^2}} \quad (4.1)$$

where x_i and y_i represent values of two performance variables, and \bar{x} and \bar{y} are their respective mean values. Our observations indicate that, for a given performance metric, the correlation between lower percentiles is generally higher than that between higher percentiles. For instance, we may notice significant differences between the 75th and 95th percentiles of delivery rate, but not between the 5th and 25th percentiles. We found that our correlation-based selection ended up including more performance statistics from higher percentiles as input when constructing GMMs.

4.1.2 Model Selection

GMMs group input performance data into a predefined number of blobs, each with distinct performance characteristics. A key challenge in probabilistic modeling is to determine the “correct” number of blobs in a GMM that best fits the input data and generalizes well to new data. This process, known as model selection, often requires external ground-truth information. When ground-truth information is unavailable, model selection can be based on intrinsic probabilis-

tic statistical metrics for evaluating the goodness of fit of the data to the model, including the Bayesian Information Criterion (BIC) (118), Akaike Information Criterion (AIC) (119), or Silhouette Score (120). These intrinsic metrics use maximum likelihood estimation (MLE) (121) to determine the probability distribution and its parameters that best explain the input data. However, we find that relying solely on MLE for model selection with RPC telemetry data is insufficient due to the large variations and significant overlaps in RPC statistics caused by the inherent diversity in HDCs. Therefore, additional information is required for assisting in model selection, as discussed below.

Model selection can be also informed by using categorical attributes associated with each RPC measurement. These attributes — such as source and destination job and user name, congestion control algorithm, and QoS priority — each characterizes a specific aspect of the RPC that can have a significant impact on its network performance. Therefore, these attributes can be used to assist in model selection for better interpretability¹. Specifically, when comparing two GMMs with similar BIC scores, it may be beneficial to select the one that demonstrates better separation of categorical attributes among different blobs. This enhances the interpretability of the model, as demonstrated in the examples later.

In summary, our model selection algorithm considers both categorical attributes and probabilistic statistical metrics, and evaluates the following three aspects: goodness of fit, goodness of separation, and goodness of interpretation. Next, we examine each of these aspects in more detail.

¹For ensuring unbiased modeling results, however, it is crucial to avoid using categorical attributes as the sole basis for model selection or as input for building GMMs. This is because RPCs with the same categorical attributes may have different performance, while those with different attributes may have similar performance. For example, the congestion control algorithm affects how RPCs utilize bandwidth resources and respond to network congestion. Larger RPCs can result in differing delivery rates and performance with different algorithms, such as CUBIC (103) and BBR (99). However, the effect of the congestion control algorithm on performance may be insignificant for small RPCs that can fit in a single packet.

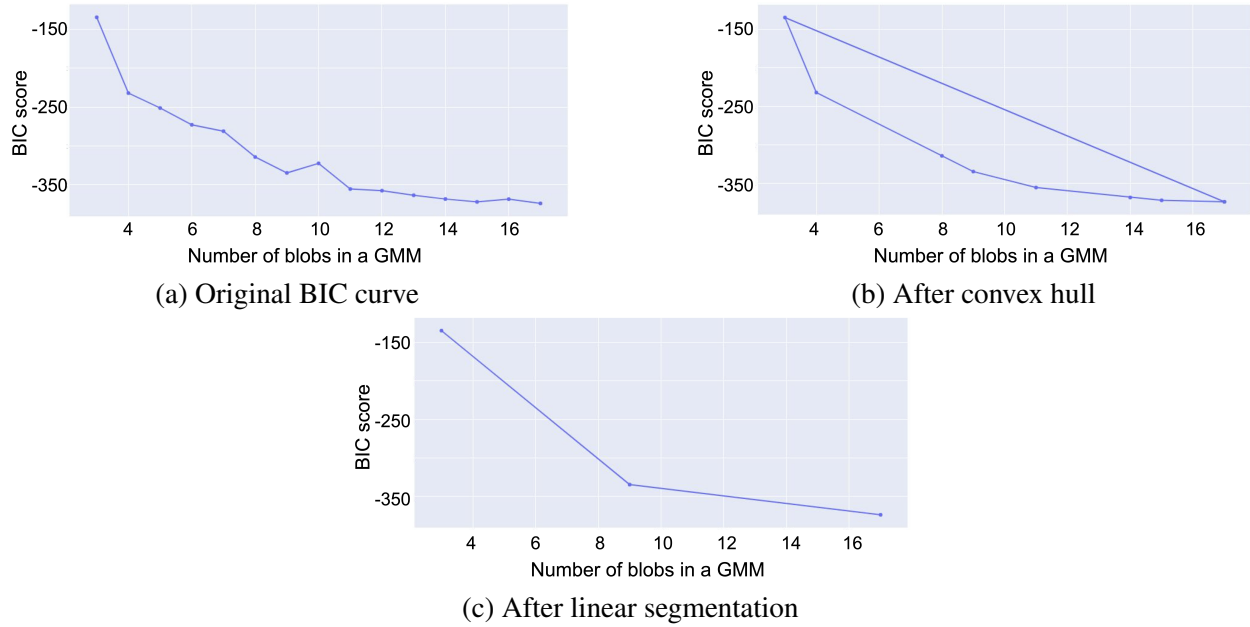


Figure 4.3: Elbow detection for an example BIC curve (x-axis: number of blobs in a GMM; y-axis: BIC score).

4.1.2.1 Goodness of Fit

Goodness of fit measures how well (the likelihood) a GMM fits the input data. Increasing the number of blobs in a GMM can improve the likelihood of the model, but it can also cause overfitting, where the model focuses on fitting the noise or random variations in the data rather than the underlying stable patterns. To balance the trade-off between fitting the data well and generalizing to new data, we use the Bayesian Information Criterion (BIC) score for evaluating GMMs with different numbers of blobs. BIC considers both the model’s likelihood given the data and its complexity, with a greater penalty for increased complexity as the number of blobs increases (122). Generally, the model with the lowest BIC score is considered the best. However, as shown in Figure 4.3(a), the decrease in BIC slows as the number of blobs increases past the “elbow” point on the curve, indicating a loss of generalizability. Therefore, models that are near the elbow on the BIC curve are preferred as they balance the trade-off between fit and complexity. Our objective is to identify the elbow point along the BIC curve.

Manual visual analysis of elbow detection can lead to subjective results, which poses a significant challenge. Furthermore, as shown in Figure 4.3(a), the BIC curve may not consistently

decrease as the number of blobs increases, making it more difficult to identify the elbow. Therefore, our approach must account for fluctuations in the BIC curve and accurately locate the elbow point, without introducing any subjectivity. To achieve this, our approach consists of two steps, as illustrated in Figure 4.3. First, we apply the convex hull method (123) to remove “inner” nodes along the curve that do not reflect the stable changing trend of the BIC score (as shown in Figure 4.3(b)). These inner nodes are typically anomalies and can be ignored. Next, we use linear segmentation to further reduce the number of nodes on the curve. Specifically, we calculate the area of each triangle formed by three consecutive nodes on the curve and iteratively eliminate the middle node of the triangle with the smallest area (Figure 4.3(c)). This approach results in three points along the BIC curve: the minimal, maximum, and middle point that indicates the elbow. Models beyond the elbow are filtered out for generalizability. In this way, we can accurately locate the elbow point, which represents the optimal number of blobs in the GMM.

4.1.2.2 Goodness of Separation

Goodness of separation measures the separability of RPC performance measurements in different blobs within a GMM. GMMs adopt “soft clustering” and assign each measurement i to a blob n with probability p_{in} , where the sum of all probabilities is 1 ($\sum_n p_{in} = 1$). We calculate the entropy of each measurement i based on the probabilities assigned by the GMM using the following equation (Eq. 4.2).

$$Entropy_i = - \sum_n p_{in} \log p_{in} \quad (4.2)$$

The entropy represents the degree of uncertainty of the measurement; a lower entropy value indicates a higher confidence that the model is accurately assigning the measurement to a single blob.

Figure 4.4 illustrates an example of two GMMs for the same input measurements, with two or three blobs. As shown in GMM_1 (Figure 4.4(a)), measurement m_a is located in the center of the upper blob. The probability vector of m_a assigned to the lower and upper blob in GMM_1



Figure 4.4: Comparison of two GMMs with different number of blobs. For goodness of separation, GMM_1 (a) is preferred.

is $\langle 0, 1 \rangle$, since these two blobs are well separated. However, as seen in Figure 4.4(b) when the number of blobs increases to three, measurement m_a can be assigned to one of the two upper blobs in GMM_2 since these two blobs are close and overlap significantly. In the worst case, the probability vector of m_a in GMM_2 is $\langle 0, 0.5, 0.5 \rangle$, resulting in an increase of entropy from 0 in GMM_1 to 0.3 ($\log 2$) in GMM_2 . Thus, GMM_1 is preferred for better separation of blobs.

To assess the separability of measurements within a GMM, we compute the entropy for each measurement and calculate the average value. When comparing multiple GMMs with different numbers of blobs, a GMM with a lower average entropy value is considered to have better separability and is preferred.

4.1.2.3 Goodness of Interpretation

Goodness of interpretation evaluates the degree to which the modeling results can be easily understood and interpreted in relation to categorical attributes. Each RPC measurement is annotated with a set of categorical attributes that distinguish it based on factors such as the applications it belongs to (e.g., users or jobs), the locations in which it runs (e.g., clusters), and the network protocols it uses (e.g., congestion control algorithms or transmission priorities). These attributes can greatly impact the network performance of RPCs in HDCs. By analyzing the attributes that are most consistent among RPCs within the same blob or most different among

RPCs in different blobs, we can gain insight into the effects of these attributes on different network performance, as seen in the modeling results.

For measuring interpretability, we use impurity analysis and focus on two main aspects: uniformity and spread. Uniformity evaluates the consistency of categorical attributes within a blob, while spread assesses the distribution of measurements with the same categorical attributes among different blobs. There is a trade-off between these two aspects: When each measurement data point forms an individual blob, it is optimal for uniformity but worst for spread; when all measurements are grouped in a single blob, it is optimal for spread but worst for uniformity. For optimal interpretability, we aim for models that group measurements with similar categorical attributes together in the same blob (uniformity) and separate measurements with different attributes in different blobs (spread). As an example, consider two GMMs constructed from measurements with three different transmission priorities: AF1, AF2, and AF3 (Figure 4.5). GMM_1 contains two blobs: one blob contains only AF3 RPCs, and the other has a combination of AF1 and AF2 RPCs (Figure 4.5(a)). GMM_2 contains three blobs, each composed of RPCs with a unique priority (Figure 4.5(b)). Compared to GMM_1 , GMM_2 is preferred because the transmission priority of RPCs in each blob is more uniform, providing clearer insight into how different transmission priorities are associated with different performance.



Figure 4.5: Example GMMs. For goodness of interpretation, GMM_2 (b) is preferred.

For evaluating the interpretability of the modeling results, we calculate the information gain (124) with categorical attributes from these two perspectives described above as ($IG_{uniform} -$

$IG_{diverse}$). $IG_{uniform}$ measures the decreases in impurity after clustering (Eq. 4.3), and $IG_{diverse}$ measures the increases in impurity after clustering (Eq. 4.4). For optimal interpretability, we want $IG_{uniform}$ to be as large as possible to maximize consistency of categorical attributes within blobs, and $IG_{diverse}$ to be as small as possible to minimize spread among different blobs. Therefore, a higher value of $(IG_{uniform} - IG_{diverse})$ is preferred.

$$IG_{uniform} = - \sum_{c_i} p(c_i) \log p(c_i) + \sum_{b_j} \frac{N_{b_j}}{N} \left\{ \sum_{c'_i \in b_j} p(c'_i) \log p(c'_i) \right\} \quad (4.3)$$

$$IG_{diverse} = - \sum_{c_i} \sum_{b'_j \in c_i} p_{b'_j} \log p_{b'_j} \quad (4.4)$$

c_i : categorical attribute i

b_j : blob j in a GMM

$c'_i \in b_j$: categorical attribute i' in blob j

$b'_j \in c_i$: blob j' contains measurements with categorical attribute i

	b_1	b_2	b_3	b_4
GMM_a	20 c_1	30 c_2	30 c_3	20 c_3
GMM_b	20 c_1 , 20 c_2	10 c_2 , 20 c_3	30 c_3	—

Table 4.1: An example of two GMMs and their categorical attributes in each blob (b_n).

Let us calculate these two metrics with an example. Consider 100 measurements with three unique categorical attributes: 20 measurements with c_1 , 30 measurements with c_2 , and 50 measurements with c_3 . We construct two GMMs (GMM_a and GMM_b) using these 100 measurements. GMM_a consists of 4 blobs, with 20 measurements having attribute c_1 in blob b_{a1} , 30 measurements having attribute c_2 in blob b_{a2} , 30 measurements having attribute c_3 in blob b_{a3} , and 20 measurements having attribute c_3 in blob b_{a4} . GMM_b consists of 3 blobs, with 20 measurements having attribute c_1 and 20 measurements having attribute c_2 in blob b_{b1} , 10 measurements having attribute c_2 and 20 measurements having attribute c_3 in blob b_{b2} , and 30 measurements having

attribute c_3 in blob b_{b_3} as shown in Table 4.1. We can calculate $IG_{uniform}$ and $IG_{diverse}$ for each GMM blob using Eq. 4.3 and Eq. 4.4:

$$\begin{aligned}
\{IG_{uniform}\}_a &= \left\{ - \sum_{c_i} p(c_i) \log p(c_i) \right\} + \left\{ \sum_{b_j} \frac{N_{b_j}}{N} \left\{ \sum_{c'_i \in b_j} p(c'_i) \log p(c'_i) \right\} \right\}_a \\
&= \left\{ -\frac{20}{100} * \log \frac{20}{100} - \frac{30}{100} * \log \frac{20}{100} - \frac{50}{100} * \log \frac{50}{100} \right\} \\
&\quad + \left\{ \left\{ \frac{20}{100} * 1 * \log 1 \right\}_{b_1} + \left\{ \frac{30}{100} * 1 * \log 1 \right\}_{b_2} + \right. \\
&\quad \left. + \left\{ \frac{30}{100} * 1 * \log 1 \right\}_{b_3} + \left\{ \frac{20}{100} * 1 * \log 1 \right\}_{b_4} \right\} \\
&\approx 0.447 + 0 = 0.447
\end{aligned}$$

$$\begin{aligned}
\{IG_{diverse}\}_a &= - \sum_{c_i} \sum_{b'_j \in c_i} p_{b'_j} \log p_{b'_j} \\
&= - \left\{ \frac{20}{100} * \left(\frac{20}{20} * 1 * \log 1 \right) \right\}_{c_1} \\
&\quad - \left\{ \frac{30}{100} * \left(\frac{30}{30} * 1 * \log 1 \right) \right\}_{c_2} \\
&\quad - \left\{ \frac{50}{100} * \left(\frac{30}{50} * \log \frac{30}{50} \right. \right. \\
&\quad \left. \left. + \frac{20}{50} * \log \frac{20}{50} \right) \right\}_{c_3} \\
&\approx 0.146
\end{aligned}$$

$$\{IG_{uniform} - IG_{diverse}\}_a = 0.447 - 0.146 = 0.301 \tag{4.5}$$

$$\begin{aligned}
\{IG_{uniform}\}_b &= \left\{ - \sum_{c_i} p(c_i) \log p(c_i) \right\} + \left\{ \sum_{b_j} \frac{N_{b_j}}{N} \left\{ \sum_{c'_i \in b_j} p(c'_i) \log p(c'_i) \right\} \right\}_b \\
&= \left\{ - \frac{20}{100} * \log \frac{20}{100} - \frac{30}{100} * \log \frac{20}{100} - \frac{50}{100} * \log \frac{50}{100} \right\} \\
&\quad + \left\{ \left\{ \frac{40}{100} * \left(\frac{20}{40} * \log \frac{20}{40} + \frac{20}{40} * \log \frac{20}{40} \right) \right\}_{b_1} \right. \\
&\quad + \left\{ \frac{30}{100} * \left(\frac{10}{30} * \log \frac{10}{30} + \frac{20}{30} * \log \frac{10}{30} \right) \right\}_{b_2} \\
&\quad \left. + \left\{ \left\{ \frac{30}{100} * 1 * \log 1 \right\}_{b_3} \right\} \right\} \\
&\approx 0.447 - 0.120 - 0.083 = 0.244
\end{aligned}$$

$$\begin{aligned}
\{IG_{diverse}\}_b &= - \sum_{c_i} \sum_{b'_j \in c_i} p_{b'_j} \log p_{b'_j} \\
&= - \left\{ \frac{20}{100} * \left(\frac{20}{20} * 1 * \log 1 \right) \right\}_{c_1} \\
&\quad - \left\{ \frac{30}{100} * \left(\frac{20}{30} * \log \frac{20}{30} + \frac{10}{30} * \log \frac{10}{30} \right) \right\}_{c_2} \\
&\quad - \left\{ \frac{50}{100} * \left(\frac{20}{50} * \log \frac{20}{50} + \frac{30}{50} * \log \frac{30}{50} \right) \right\}_{c_3} \\
&\approx 0.083 + 0.146 = 0.229
\end{aligned}$$

$$\{IG_{uniform} - IG_{diverse}\}_b = 0.244 - 0.229 = 0.015 \tag{4.6}$$

Based on the values of $(IG_{uniform} - IG_{diverse})$ calculated for these two models, GMM_a is determined to have better interpretability and is preferred due to its higher $(IG_{uniform} - IG_{diverse})$ value.

Next, we consider an example from a production HDC. In this example, we want to decide which model better interprets the network performance of RPCs for an application that stores larger binary data in a cluster of production HDCs. Before evaluating the interpretability of the

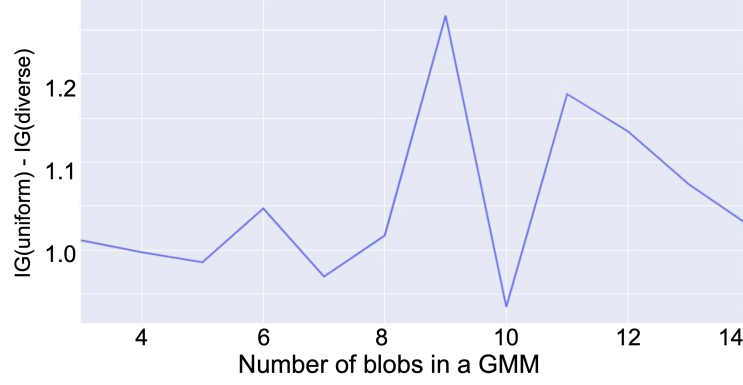


Figure 4.6: An example of goodness of interpretation analysis with RPC measurements in a production HDC.

models, we filter out models with more than 8 blobs, as they tend to lack generalization based on the BIC curve analysis. As shown in Figure 4.6, among models with no more than 8 blobs, the GMM with 6 blobs has the highest information gain ($(IG_{uniform} - IG_{diverse})$) and is therefore selected as the preferred model. To better understand the trend of information gain in Figure 4.6, we analyze the categorical attributes of each blob in GMMs with 5, 6, or 7 blobs. Figure 4.7 shows the 2D distribution of the pacing rate and the delivery rate of RPC measurements at the 95th percentile for GMMs with 5 (GMM_5 : left), 6 (GMM_6 : middle), or 7 (GMM_7 : right) blobs. As the number of blobs increases from 5 to 6, blob C in GMM_5 is split into two blobs, C_1 and C_2 in GMM_6 . The results of the categorical analysis indicate that RPCs in blob C in GMM_5 are related to three main destination users: $u1$, $u2$, and $u3$. Table 4.2 shows the percentage of RPCs attributed to each destination user in blob C in GMM_5 , and in C_1 and C_2 in GMM_6 . The results show that in GMM_6 the RPCs related to $u3$ are separated from the other two users and grouped in blob C_1 , leading to a more uniform categorical composition in blobs C_1 and C_2 , thus improving interpretability. Comparing the information gain of RPCs in blob C in GMM_5 to blobs C_1 and C_2 in GMM_6 , $\{IG_{uniform}\}_5 - \{IG_{diverse}\}_5 = 0 - 0 = 0$ and $\{IG_{uniform}\}_6 - \{IG_{diverse}\}_6 = 0.26 - 0 = 0.26$. The calculations suggest that GMM_6 is preferred, as indicated by the higher value of $(\{IG_{uniform}\} - \{IG_{diverse}\})_6$ and is consistent with the categorical analysis. As the number of blobs increases to 7 in GMM_7 , RPCs in blobs C_1 and C_2 merge into a single blob again, similar to that in GMM_5 , resulting in a decrease in information gain.

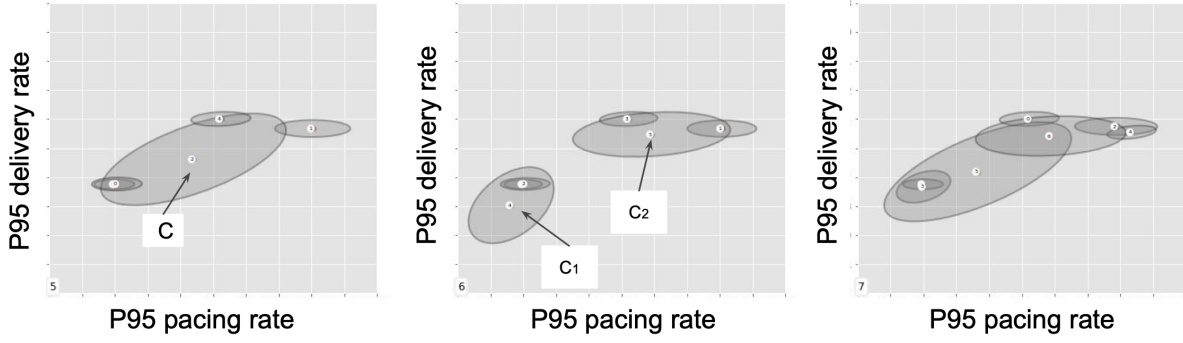


Figure 4.7: An example of model selection with a production dataset: the 2D distribution of 95th percentile pacing rate (x -axis) and delivery rate (y -axis) when the number of blobs in GMM is 5 (left), 6 (middle), and 7 (right) respectively (actual values are proprietary and hidden).

<i>blob</i>	<i>dst_user</i>	<i>ratio</i>
C	u_1	39.63%
	u_2	30.73%
	u_3	29.63%

<i>blob</i>	<i>dst_user</i>	<i>ratio</i>
C_1	u_3	100%
C_2	u_1	51.01%
	u_2	48.99%

Table 4.2: Percentages of RPCs for different destination users in blobs C , C_1 , and C_2 .

4.1.2.4 Model Selection Approach Summary

Model selection can be a difficult task without external ground-truth information as a reference. To overcome this limitation, our model selection approach evaluates the goodness of fit, the goodness of separation, and the goodness of interpretation for GMMs with different numbers of blobs, based on both probabilistic statistical metrics and categorical attributes for the given RPC telemetry data. Specifically, we follow the following steps for model selection:

1. First, we evaluate the *goodness of fit* of the models using the BIC score and eliminate models that fall beyond the elbow of the BIC curve. This ensures a good balance between model fitness and generalization.
2. Next, we select the top N models that demonstrate better *separation* based on the average entropy calculated from the clustering probability.

3. Lastly, among the top N models, we choose the one that has the most informative *interpretation*, as determined by the highest information gain calculated from the categorical attributes.

We have evaluated our model selection approach using several real-world case studies in a production HDC. Below, we present one such example of these evaluations. As we do not have ground-truth information, our approach for assessing the performance of our model selection mainly relies on categorical analysis and feedback from domain experts. In this specific example, we model the network performance of a storage service in one cluster of the production HDC using GMMs. For model selection, we incrementally increase the number of blobs in the GMM from 3 to 14, and apply our approach to determine the best model.

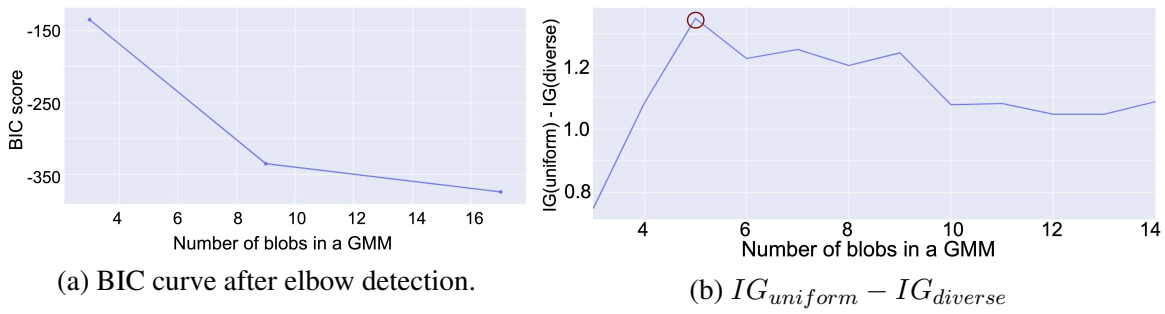


Figure 4.8: The results of elbow detection and impurity analysis in example 1. The number of blobs in GMMs ranges from 3 to 17.

- **Goodness of Fitness** Figure 4.8(a) shows the BIC curve after applying the elbow detection algorithm to the original curve (Figure 4.3(a)). The results indicate that, as the number of blobs increases beyond 9, the model's ability to generalize to unseen data decreases.
- **Goodness of Separation** Based on the clustering probability, GMMs with 4, 3, 5, 7, and 8 blobs have the lowest entropy. Figure 4.9 shows the 2D distribution of the 50th percentile of congestion window size (x -axis) and the 95th percentile of delivery rate (y -axis) for GMMs with 4 (*left*), 5 (*middle*), or 6 (*right*) blobs. The figure illustrates that when the number of blobs increases from 5 (*middle*) to 6 (*right*), blob B is divided into two overlapping blobs (B_1 and B_2), resulting in an increase in entropy.

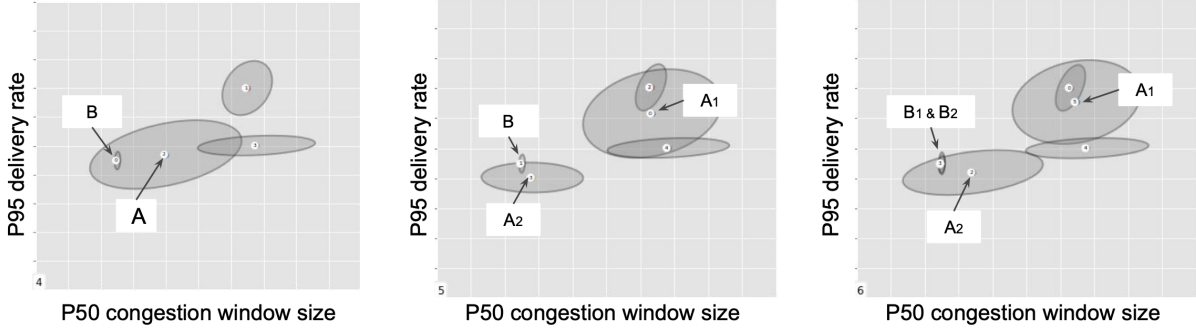


Figure 4.9: An example of model selection with a production dataset: the 2D distribution of the median congestion window (x -axis) and the 95th percentile delivery rate (y -axis) when the number of blobs in GMM is 4 (*left*), 5 (*middle*), and 6 (*right*) respectively (actual values are proprietary and hidden).

- **Goodness of Interpretation** Figure 4.8(b) shows the values of $(IG_{uniform} - IG_{diverse})$ for GMMs with different numbers of blobs. The model with 5 blobs has the highest value, indicating a better interpretation in terms of categorical attributes. Specifically, when the number of blobs increases from 4 to 5, blob A is divided into two blobs, A_1 and A_2 , as shown in Figure 4.9. A closer examination of the categorical attributes, as presented in Table ??, reveals that the measurements in A_1 and A_2 are from different destination users. Therefore, the division of the measurements in A into A_1 and A_2 leads to a more uniform categorical composition and better interpretation in each blob. However, when the number of blobs increases to 6, the measurements of destination user u_4 are moved from blob B to two overlapping blobs (B_1 and B_2), which leads to a larger increase in impurity and a smaller information gain.

Based on the aforementioned evaluation criteria, our modeling selection approach selects the GMM with 5 blobs, as depicted in the middle of Figure 4.9. This model provides the best balance between goodness of fit, goodness of separation, and goodness of interpretation.

4.1.3 Background Removal

Noise and outliers are inevitable when tracing performance data in production HDCs, as no tracing system can be completely fault-free. These faulty measurements can greatly affect the

<i>blob</i>	<i>dst_user</i>	<i>ratio</i>
<i>A</i>	<i>u</i> ₁	59.38%
	<i>u</i> ₂	27.43%
	<i>u</i> ₃	13.19%
<i>B</i>	<i>u</i> ₄	100%

<i>blob</i>	<i>dst_user</i>	<i>ratio</i>
<i>A</i> ₁	<i>u</i> ₂	82.92%
	<i>u</i> ₃	17.08%
<i>A</i> ₂	<i>u</i> ₁	100%
<i>B</i> ₁ / <i>B</i>	<i>u</i> ₄	100%
<i>B</i> ₂ / <i>B</i>		

Table 4.3: Percentages of RPC measurements for different destination users in blobs (*A*, *A*₁, *A*₂, *B*, *B*₁ and *B*₂) in Figure 4.9.

modeling results when using GMMs, as GMMs attempt to assign *every* measurement, including noise and outliers, to one of the specified blobs. This can result in distorted means and covariances, negatively impacting the usefulness of the modeling results by obscuring actual patterns and structures in the input data. Figure 4.10 illustrates an example of how an outlier can distort mean and covariance in a modeling result. In this case, the GMM has two blobs, *A* and *B*, each representing a different performance. When an outlier is introduced to the input data without changing the number of blobs in the GMM, the outlier must be assigned to one of the two blobs, either *A* or *B*. As a result, the mean and variation of the distribution represented by the measurements in whichever blob includes the outlier are falsely altered. To mitigate this issue, it is necessary to filter out noise and outliers in the performance data. In this section, we discuss methods for doing so.

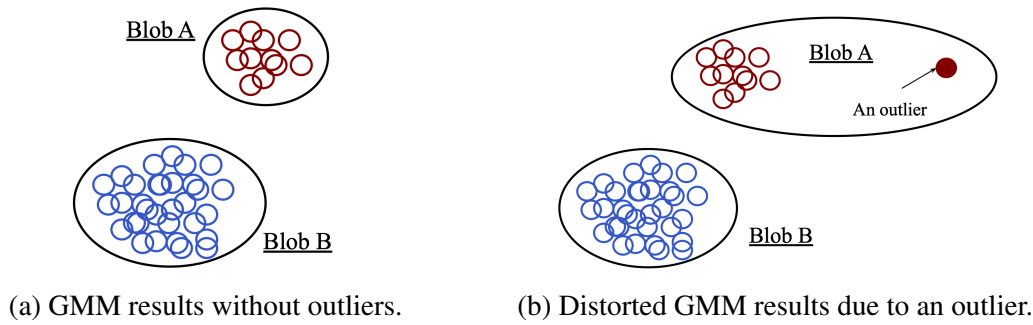


Figure 4.10: An example of how an outlier distorts the GMM result. The inner small circles illustrate the scatter plots of measurements in a 2D dimension, with the outer circles representing the contour plot of the 2D distribution for measurements in each blob.

It is important to note that the noise/outliers we aim to remove in this context are distinct from performance anomalies in HDCs. Performance anomalies are caused by insufficient re-

sources, unexpected outages, or misconfigurations and accurately reflect the experience of RPCs in HDCs. Therefore, measurements related to performance anomalies should be retained and identified, as these provide valuable information for performance assessment and troubleshooting. On the other hand, noise/outliers are often caused by unexpected data corruptions that occur randomly during performance tracing and do not accurately represent the experience of RPCs in HDCs. It is necessary to implement methods that can distinguish between noise/outliers and performance anomalies, in order to filter out the former without removing any meaningful measurements associated with the latter.

One common approach for filtering out noise/outliers is to use general statistical measures derived from performance data, such as the mean (μ) and standard deviation (α). For example, measurements that fall outside the $(\mu \pm 4\alpha)$ interval can be considered as potential noise/outliers and removed from the input data before GMM modeling. However, this method is not effective at distinguishing between noise/outliers and performance anomalies, as both can significantly deviate from normal performance. Additionally, using static thresholds based on overall statistics can mistakenly remove measurements related to normal behaviors in certain circumstances. This is because applications can experience different performance based on factors such as service type, workload, hardware configuration, and temporal and geographical impacts. For example, latency experienced by bandwidth-intensive and computation-intensive applications on remote hosts may differ by a few milliseconds to tens of seconds. If there are many more RPCs from bandwidth-intensive applications in HDCs than from computation-intensive ones, the average delay on remote hosts will be determined by the former. As a result, measurements of computation-intensive applications with a higher remote delay may be mistakenly filtered out based on overall statistics. Besides, these thresholds may need to be constantly updated as HDCs and applications evolve, which can be time-consuming and resource-intensive.

4.1.3.1 Our Approach

Our approach for eliminating noises/outliers in performance data is based on two observations from our experience. First, although noise/outliers and performance anomalies both deviate significantly from normal performance, they have distinct characteristics. Specifically, performance issues caused by a specific factor are likely to consistently impact the relevant RPCs, while noise/outliers introduced by random data corruption may have very different values. In a multidimensional performance space, measurements associated with a specific performance anomaly tend to aggregate, forming a discernible pattern. Conversely, measurements associated with noise/outliers tend to be dispersed, exhibiting significant variations in their values. Second, in a well-designed tracing system in production HDCs, the number of noise/outliers is expected to be lower compared to performance anomalies. Nonetheless, the number of measurements in both cases should still be significantly smaller than the number of measurements that represent normal performance.

Due to the high variability and limited quantity of noise/outliers, it is likely that noise/outliers will be mixed with normal measurements in GMM blobs, instead of being separated into individual blobs, unless the number of blobs in the GMM is greatly increased. In such cases, normal measurements with similar performance characteristics may also be separated into multiple GMM blobs, making it challenging to differentiate between blobs formed by normal measurements and those formed by noise/outliers. Our model selection approach tackles this challenge by preventing such cases, as these models have poor fitness, separability, and interpretability and are always rejected. In a GMM, blobs affected by widely dispersed noise/outliers typically exhibit significant variations across one or more performance dimensions. On the other hand, blobs composed exclusively of normal measurements (or performance anomalies) generally display densely clustering with smaller variations.

Our approach for distinguishing between noise/outliers and performance anomalies involves using GMMs. The process begins with constructing a GMM without discarding any measurements, then identifying blobs that may contain noise/outliers, referred to as background blobs.

The final step is removing the noise/outliers from these background blobs and reconstructing the GMM.

To differentiate background blobs from others, we use the fact that noise/outliers result in larger variations across one or more performance dimensions in GMM blobs. However, we avoid setting a static threshold for variations when identifying background blobs as the variations of normal measurements across different performance dimensions can vary greatly and noise/outliers are inherently unpredictable, making it challenging to determine an appropriate threshold. To address this challenge, we use the classical Mahalanobis distance (125) as a measure of dissimilarity between two multidimensional blobs, A and B, to distinguish between background blobs and other blobs. The distance between a measurement in B and the distribution of A is calculated using Eq. 4.7:

$$d_M(\vec{m}, D)^2 = (\vec{m} - \vec{\mu})^T S^{-1} (\vec{m} - \vec{\mu}) \quad (4.7)$$

where N is the number of performance dimensions, $\vec{m} = (m_1, m_2, m_3, \dots, m_N)^T$ represents a measurement in blob B ($M(B)$), $\vec{\mu} = (\mu_1, \mu_2, \mu_3, \dots, \mu_N)^T$ is the mean vector of blob A ($D(A)$), and S is the covariance matrix. The distance represents the number of standard deviations that a measurement in blob B is from the mean of the distribution of blob A. By comparing the average distance between the measurements and the distributions of any two blobs, we can distinguish between background and non-background blobs. Typically, non-background blobs have smaller variations and differ significantly from each other in at least one performance dimension, resulting in small, symmetric distances between their distributions and measurements. That is, the distance between distribution $D(A)$ and measurement $M(B)$ is similar to the distance between distribution $D(B)$ and measurement $M(A)$ when blobs A and B are non-background blobs. However, if one of the blobs is a background blob, their distances become asymmetric. Specifically, the distance between the distribution of background blobs and measurements in non-background blobs is small, given the larger variations of background blobs, while the distance between the

distributions of non-background blobs and measurements in background blobs is large, as measurements in background blobs can scatter widely. This allows us to identify background blobs using the Mahalanobis distance.

It should be noted that not all measurements in the background blob(s) can be considered noise/outliers. After identifying the background blob(s), we can use *leverage* (126), a measure of how far away the variable values of a measurement are from those of the other measurements, to identify and remove potential noise/outliers and rebuild a GMM. High-leverage measurements are often considered outliers and can cause large changes in the parameter estimates when they are removed. This aligns with our observations that noise/outliers can significantly impact the results of a GMM. For identifying high-leverage measurements in the background blob(s), we calculate the Mahalanobis distances between each measurement in the blob(s) and the blob distribution (127). If the distance is greater than a threshold, the measurement is considered an outlier and removed. Finding a static threshold that effectively covers all background blobs is difficult because of the unpredictable behavior of noise/outliers and their tendency to have varying values. To determine the threshold for a background blob i , we compute the ratio of the average distance between measurement $M(i)$ and distribution $D(j)$, where j is a non-background blob, to the average distance between measurement $M(j)$ and distribution $D(i)$ (Eq. 4.8). In cases where multiple non-background blobs exist, the minimum ratio is employed as the threshold for filtering out as many high-leverage measurements as possible from the background blob (Eq. 4.9).

$$threshold_{(i,j)} = d_M(\overrightarrow{m_{bg(i)}}, D_{non-bg(j)}) / d_M(\overrightarrow{m_{non-bg(j)}}, D_{bg(i)}) \quad (4.8)$$

$$threshold_i = \min_j(threshold_{(i,j)}) \quad (4.9)$$

4.1.3.2 Validation with a Public Dataset

We apply our approach to a public Yahoo dataset, which is commonly used for anomaly detection (128) — we refer to the dataset as Y_1 . Figure 4.11 shows the time series plot of the

95th percentile of latency in Y_1 . The blue points represent normal measurements, which follow a normal distribution $\mathcal{N}(1, 0.25)$, and the orange points are random noise injected into the dataset. There are a total of 63,121 normal measurements and 11,535 injected noise points.

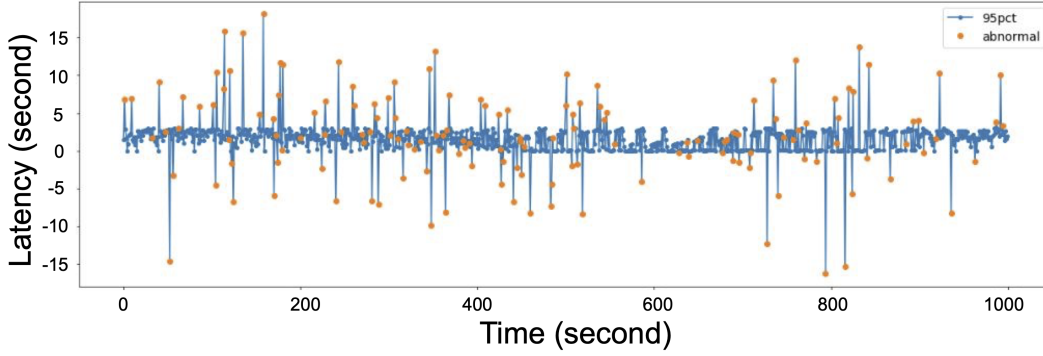


Figure 4.11: Time series plot of latency in 95th percentile from Y_1 .

We construct a GMM with 2 blobs using these measurements. The probability distribution of latency for each blob is shown in Figure 4.12(a). According to ground truth information, blob 1 mainly contains normal measurements, while blob 2 contains mostly noise/outliers (10,381 out of 11,535). The Mahalanobis distance between the distributions and measurements within/between different blobs is presented in Table 4.4. The distance between the measurements in blob 1 ($M(1)$) and the distribution of blob 2 ($D(2)$) is around 0.32, while the distance between the measurements in blob 2 ($M(2)$) and the distribution of blob 1 ($D(1)$) is 1,273. Therefore, blob 2 is considered a background blob affected by noise/outliers. After removing noise/outliers identified using our approach in blob 2, the probability distribution of latency for the input data is shown in Figure 4.12(b). The resulting distribution closely matches the ground-truth distribution $\mathcal{N}(1, 0.25)$.

	$M(1)$	$M(2)$
$D(1)$	4.99	1273.75
$D(2)$	0.32	4.99

Table 4.4: The Mahalanobis distance within/between blobs in Figure 4.12(a).

We next modify the Yahoo dataset by adding 63,121 measurements that follow a normal distribution $\mathcal{N}(11, 0.25)$. These measurements are considered normal. We refer to this modified

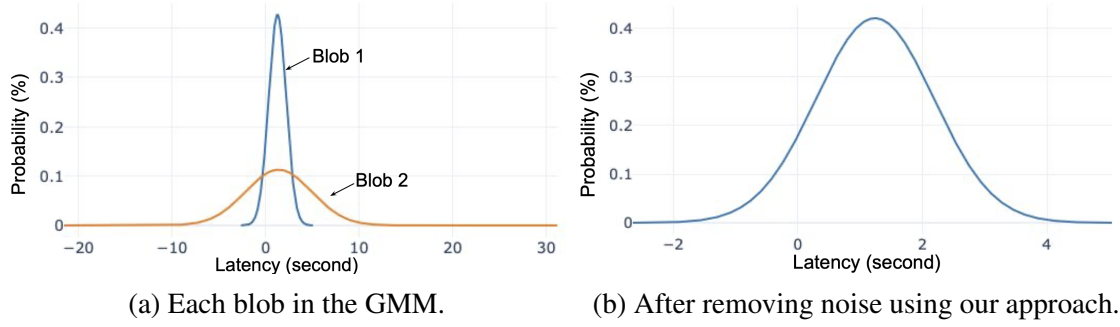


Figure 4.12: Probability distributions of latency from Y_1 in Figure 4.11.

dataset as Y_2 . Our goal is to evaluate whether our approach can identify background blob(s) when there are multiple normal performances. Figure 4.13 shows the time series plot of the latency in the 95th percentile of Y_2 . Based on ground truth information, the normal measurements in Y_2 are drawn from two distinct Gaussian distributions. To assess the effectiveness of our background removal approach, we conduct evaluations using GMMs with 2 and 3 blobs, respectively.

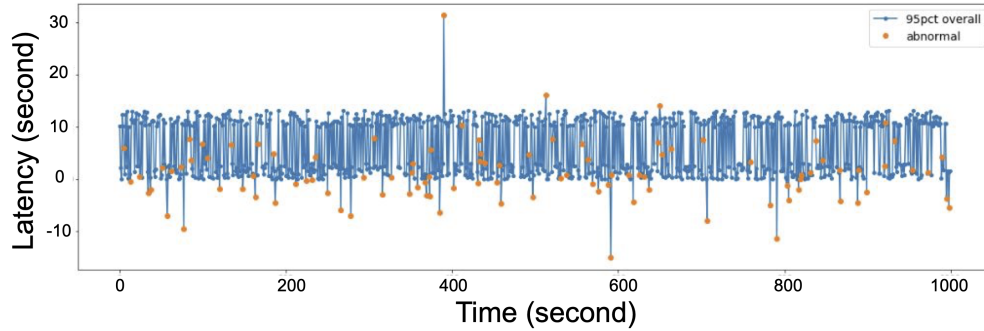


Figure 4.13: Time series plot of latency in 95th percentile in the modified Yahoo dataset.

Figure 4.14(a) displays the probability distribution of each blob in the GMM with 3 blobs. Based on ground-truth information, blob 1 contains 90% noise, while blobs 2 and 3 are mainly made up of normal measurements that follow two normal distributions. Table 4.5 presents the Mahalanobis distances among these three blobs. The measurements in blob 1 have a large distance to the distribution of blobs 2 and 3 (951 and 948, respectively), while the measurements in blobs 2 and 3 have small distances to the distribution of blob 1 (2.32 and 0.62, respectively). Meanwhile, the distances between the measurements and the distributions between blobs 2 and 3 are similar (134 vs. 130). Therefore, blob 1 is considered a background blob affected

by noise/outliers. After removing identified outliers from blob 1, the probability distribution of latency for each blob in the new GMM is shown in Figure 4.14. This clearly illustrates two different performances that are consistent with the input. In total, our approach removed 9,651 of 11,535 outliers from the input data.

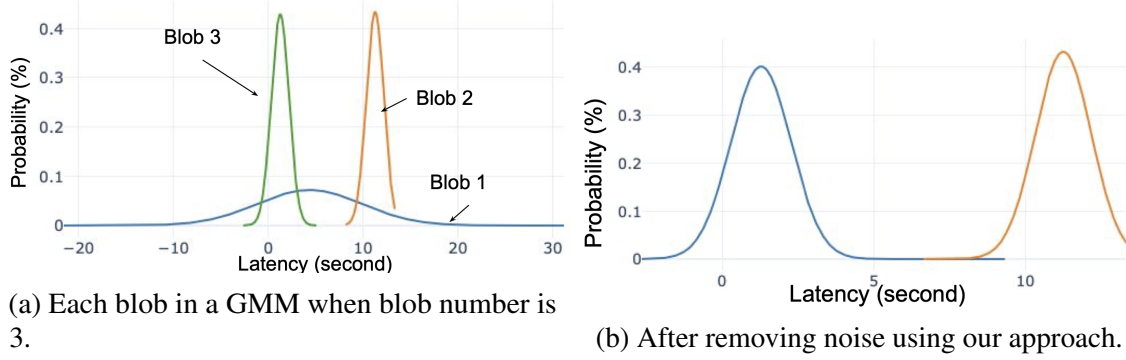


Figure 4.14: Probability distribution of latency in the modified Yahoo data set in Figure 4.13.

	$M(1)$	$M(2)$	$M(3)$
$D(1)$	4.99	2.32	0.62
$D(2)$	951.07	4.99	134.13
$D(3)$	948.04	130.39	4.99

Table 4.5: The Mahalanobis distance within/between blobs Figure 4.14(a).

We next remove noise/outliers after modeling a GMM with 2 blobs. Figure 4.15(a) shows the probability distributions of latency for each blob, and Table 4.6 displays their Mahalanobis distances. Blob 1 contains most of the injected noise and is successfully identified by our approach, as demonstrated by the smaller distance between distribution $D(1)$ and measurement $M(2)$ compared to the distance between distribution $D(2)$ and measurement $M(1)$. Figure 4.14(b) shows the probability distribution of latency for each blob in the new GMM after removing the identified noise. In total, 9,198 injected noise points are removed, revealing the actual structures in the modified Yahoo dataset.

	$M(1)$	$M(2)$
$D(1)$	4.99	38.01
$D(2)$	254.76	4.99

Table 4.6: The Mahalanobis distance between blobs in Figure 4.15(a).

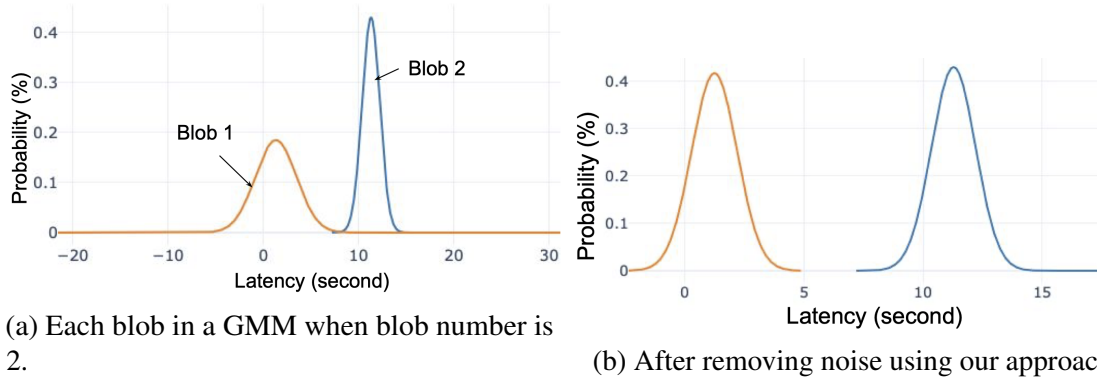


Figure 4.15: Probability distribution of latency in the modified Yahoo data set in Figure 4.13.

In this section, we present three key techniques for ensuring the accuracy of the GMM results in our modeling pipeline: feature selection, model selection, and background removal. Through evaluations using performance data from both production HDCs and public datasets, we have demonstrated the effectiveness of our approach in achieving these tasks. In the next section, we will explore the use of statistical tools to extract vital information from GMMs for effectively interpreting the network performance associated with each individual blob.

4.2 Analysis of GMM blobs

To effectively use GMMs for troubleshooting and decision-making, additional analysis is necessary for interpreting the network performance associated with each identified blob. This step involves using statistical tools to extract meaningful information from the GMM, based on the weight, mean, and covariance matrix of each multivariate Gaussian distribution of each blob. This information can aid in understanding the performance of each blob and can inform future actions. In this section, we present the statistical tools used for obtaining this information.

GMM analysis consists of three key elements: performance characterization, categorical composition analysis, and blob matching. Performance characterization identifies the most distinctive performance metrics for each blob, which can be used for understanding potential constraints that may affect performance. Categorical composition analysis highlights the shared categorical attributes among measurements within each blob and the unique categorical attributes across

different blobs. This helps identify which services or jobs are constrained and what factors, such as location or Quality of Service (QoS), may have contributed to these constraints. Blob matching ensures an “apples-to-apples” comparison when comparing multiple GMMs. This process helps to evaluate the impact of certain engineering changes on application network performance by making sure that the blobs being compared from before and after the change are equivalent. These three components work together to extract essential information from GMMs for interpreting the network performance of RPCs in HDCs. This information can assist in assessment, planning, and troubleshooting tasks. In the following section, we will provide a detailed explanation of the techniques used for each component.

4.2.1 Performance Characterization

By identifying distinguishing performance metrics for each blob in a GMM, we can determine their unique performance characteristics. These metrics, which correspond to different subsystems, provide insight into the underlying constraints that can affect performance. For example, a blob with a high delay on the client side may indicate a local constraint, while a blob with low throughput and high network latency may suggest network congestion.

To determine the most distinguishing performance metrics for each blob, we calculate the distance between the blobs along each performance metrics using the first Wasserstein distance (129). The Wasserstein distance, also known as the earth mover’s distance, measures the minimal cost required to transform one probability distribution into another. Given two distributions u and v and their cumulative distribution functions U and V , the first Wasserstein distance between u and v is defined as follows (Eq. 4.10) (130):

$$W_1(u, v) = \int_{-\infty}^{\infty} |U - V| \quad (4.10)$$

The per-metric relative distances between the blobs are then used to rank the most distinguishing performance metrics. These metrics help to understand the performance characteristics of

RPC measurements within each blob and to identify the underlying constraints. For example, if latency on the remote end is the most distinctive metric between two GMM blobs, it may suggest that the performance of RPCs in the blob with higher latency is more constrained by remote CPU/mem resources. If network delay is the most distinguishing metric, it indicates that the RPCs in the blob with higher network delay are more constrained by bandwidth resources. Concrete examples of this process can be found in Chapter 5, where we demonstrate the accuracy of GMMs in modeling the network performance of applications on a controlled testbed.

4.2.2 Categorical Composition Analysis

After characterizing the distinctive performance metrics for each blob in a GMM, we next analyze the common categorical attributes among the measurements within each blob and the unique categorical attributes across different blobs. This enables us to understand how specific categorical attributes, such as service or job types, may impact performance. This information is essential for understanding the cause of network performance associated with each blob in a GMM, and for making informed decisions on how to address any issues that may affect performance. For example, if we observe that all RPC measurements for an application X are constrained by CPU/memory resources, leading to higher latency on local and remote hosts, we can conclude that increasing the CPU and memory resources available to the application may improve its performance. Similarly, if all measurements from cluster A to cluster B in a data center are constrained by bandwidth resources, resulting in a high packet loss rate and network latency, it may indicate a connectivity issue between these two clusters, and it could be beneficial to investigate the network infrastructure between them.

For obtaining this information, we analyze the categorical composition of each blob in a GMM. Recall that categorical attributes include job-related information, such as the source and destination users and jobs, topological location information, such as source and destination pods, clusters and campuses, and transport policies such as QoS priorities and congestion control algorithms. Our goal is to answer the following questions: (1) What is the degree of diversity in

representation of a specific category within a blob? (2) How does the composition of categories within a blob differ from that of other blobs? To achieve the goal, we construct a per-category *composition vector* that represents the fraction of measurements in each blob attributed to each category. As an example, consider the QoS class category. Suppose that blob A is composed of 1,000 measurements from QoS class 2 and 6,000 measurements from QoS class 3, while blob B is composed of 100 measurements from QoS class 1 and 200 measurements from QoS class 3. The QoS class composition vectors for blobs A and B would be $[0, \frac{1}{7}, \frac{6}{7}]$ and $[\frac{1}{3}, 0, \frac{2}{3}]$, respectively. We use fractions instead of absolute numbers because the total number of measurements for each instance may differ significantly due to workload differences. For comparing the diversity of the categorical composition within a blob, we calculate the entropy (131) of the categorical composition vector for each blob based on Eq. 4.2. A low entropy value indicates high homogeneity, and ranking attributes based on entropy allows us to identify shared and unique attributes in each blob. For quantifying the similarity and difference in categorical composition across different blobs, we compute the cosine similarity (132) between the composition vectors of a given pair of blobs using Eq. 4.11:

$$\cos(A, B)_{\chi} = \frac{A \cdot B}{\|A\| \times \|B\|} \quad (4.11)$$

where A and B are composition vectors of two blobs for attribute χ , $A \cdot B$ is the dot product of these two vectors, and $\|A\|$ and $\|B\|$ are the lengths of each vector. The cosine similarity ranges from 0 to 1, with values closer to 1 indicating a greater degree of similarity. Ranking the categorical attributes based on cosine similarity reveals shared and unique attributes across different blobs. The entropy and cosine similarity analysis provides insight into the potential factors that contribute to similar or different performance among the GMM blobs. More examples will be presented through case studies in Chapter 6.

4.2.3 Blob Matching

When comparing multiple GMMs, such as those before and after an infrastructure change, it is important to ensure that the blobs being compared are matched for a fair and accurate “apples-to-apples” comparison. For example, RPCs from a MapReduce application (11) may experience diverse performance at the “map” and “reduce” stage, as the former is bandwidth-intensive and the latter is computationally-intensive. When evaluating the impact of a change in HDCs on MapReduce applications, it would be misleading to compare the performance of the “map” stage before the change with the “reduce” stage after the change, or vice versa.

To ensure accurate comparisons, we use two measures for matching the corresponding blobs among multiple GMMs:

1. Cosine similarity between categorical compositions. Matching blobs are expected to include similar users/jobs from similar QoS classes, resulting in a similar *categorical* composition and a higher measure of cosine similarity.
2. The 2-Wasserstein distance between distributions of performance measures (129; 133).

This distance is based on the mean (m) and the covariance matrix (Σ) of the probability distribution for each blob constructed by the GMM, and has a closed-form expression:

$$d^2 = \|m_1 - m_2\|_2^2 + \text{Tr}(\Sigma_1 + \Sigma_2 - 2(\Sigma_1^{1/2}\Sigma_2\Sigma_1^{1/2})^{1/2}) \quad (4.12)$$

The assumption is that RPCs that are limited by the same constraint and experience similar performance are likely to respond consistently to a planned change or a performance outage. Therefore, their before and after distributions should be more *similar* compared to others and have a smaller distance.

The matching process relies primarily on the 2-Wasserstein distance between distributions, with a secondary focus on categorical attributes using cosine similarity. It has been effective in identifying similar blobs without the need for ground-truth information, as shown through con-

trolled experiments on a testbed and in real-world case studies in a production HDC, as discussed in later chapters.

4.3 Conclusion

In this chapter, we present a comprehensive pipeline for modeling and analyzing the network performance of applications using GMMs based on RPC telemetry data. The modeling pipeline carefully selects input performance metrics for building GMMs, evaluates the goodness of fit, goodness of separation, and goodness of interpretation for selecting the model with the optimal number of blobs, and removes noise and outliers from the input data. The analysis pipeline uses statistical techniques to extract unique performance metrics and categorical attributes of each blob from the GMM results, and to ensure fair comparison among multiple GMMs. In the following chapters, we will demonstrate the effectiveness of GMM-based analysis in supporting assessment, planning, and troubleshooting tasks through case studies conducted in a controlled testbed and in a production HDC.

CHAPTER 5: EVALUATIONS ON THE CLOUDLAB EMULATION PLATFORM

This chapter evaluates the effectiveness of using GMMs for modeling and understanding the network performance of applications using an experimental testbed. To accomplish this, we conduct controlled experiments on the CloudLab emulation platform (51) and collect detailed performance data. CloudLab allows us to manage network topologies and traffic patterns, and provides visibility all the way down to the bare metal. For emulating conditions commonly found in data centers, we construct a sub-topology and traffic patterns based on those reported in previous studies (52). Using the controlled experimentation on Cloudlab, we evaluate the efficacy of GMMs in identifying different network performance behaviors experienced by RPCs and compare the modeling results with ground-truth data collected during the experiments.

This chapter is organized as follows. In Section 5.1, we introduce the experimental configuration, including the traffic characteristics and testbed topology. In Section 5.2, we describe the monitoring tools used for collecting performance data on the testbed. In Sections 5.3 to 5.5, we present our experiments and the results of applying GMMs in detail.

5.1 Experimental Methodology

5.1.1 Traffic Characteristics

To emulate data center traffic on CloudLab, we use the flow size and inter-arrival time distributions of four representative applications in the Facebook data center as reported in (52): cache follower, cache leader, web, and hadoop.¹ Our experiments focus on the cache follower applica-

¹In Facebook’s datacenters (52), web servers (web) serve web traffic; query results are stored temporarily in cache servers (cache) – including cache leaders, which handle cache coherency, and cache followers, which serve most read requests (134); and Hadoop servers (hadoop) handle offline analysis and data mining.

tion, as it has the highest volume of traffic according to its flow size distribution, as illustrated in Figure 5.1(a). This makes its traffic more likely to encounter different types of performance constraints that can lead to diverse network performance. Therefore, we use cache follower traffic to assess the ability of GMMs to capture different network performance and identify their constraints.

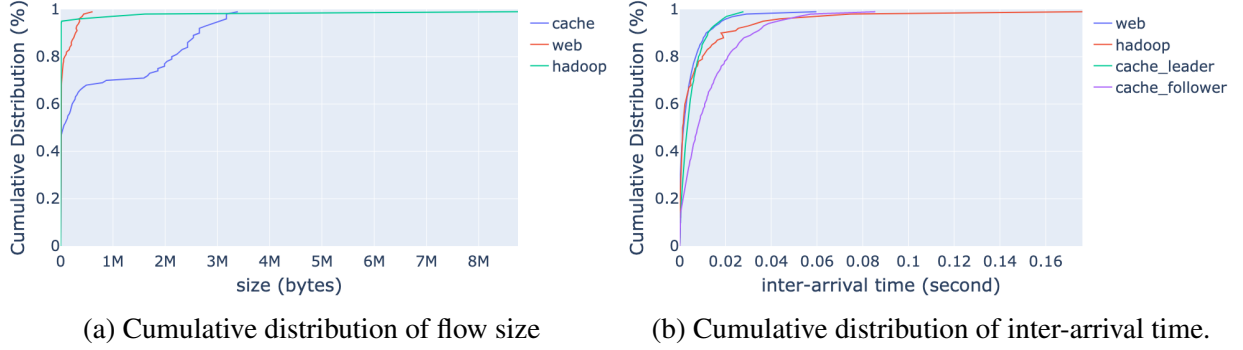


Figure 5.1: Cumulative distributions (%) of flow size and inter-arrival time of four representative applications reported in Facebook’s data centers (52). Note that *cache* in Figure 5.1(a) includes both *cache leader* and *cache follower* since they have a similar size distribution.²

5.1.2 Testbed Topology

In our experiments, we use a 3-stage Clos network topology, as described in (135) and depicted in Figure 5.2. Due to limitations in the number of switches available on CloudLab, we reduced the total number of switches to 5, including 4 aggregation switches ($Aggr_i$), each connected to 7 machines, and one core switch (*core*) that connects all the aggregation switches. All links in the testbed have 10 Gbps capacity, and the minimum RTT between a server and a client is approximately 0.05 ms, as measured using the *ping* command in Linux.

In Figure 5.2, the 14 machines under $Aggr_1$ and $Aggr_2$ (S_1 to S_{14}) are configured as servers using the open-source web server Nginx (136), while the 14 machines under $Aggr_3$ and $Aggr_4$ (C_1 to C_{14}) are used as clients. Clients use the *wget* command to send HTTP requests to servers. HTTP is a request/response protocol that is similar to RPC in its operations. The command spec-

²Curves on the plots are assembled from Figures 6 and 14 in (52).

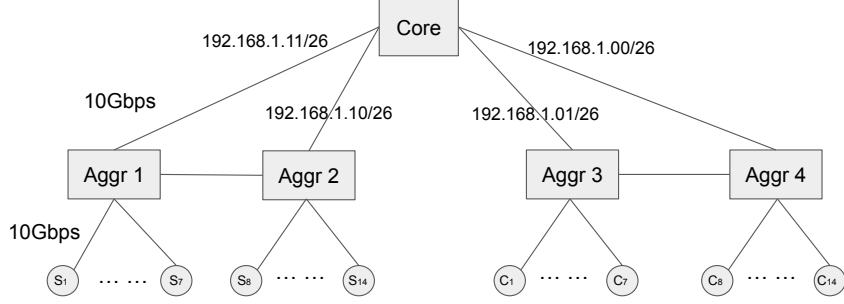


Figure 5.2: Network topology on CloudLab.

ifies the server’s IP address, port number, and the path to the requested file (e.g., `wget -q -O /dev/null http://192.168.1.11:80/downloads/cache/dummy_3950`). The number in the file name (e.g., 3950) represents the size of the file in bytes. This allows the clients to control the flow size distributions to match Figure 5.1(a) during experiments. By controlling the time gap between consecutive `wget` commands, we can ensure that the inter-arrival time distribution aligns with Figure 5.1(b).

Note that in this chapter, the generic term RPC is used to represent the request/response HTTP protocol implemented in `wget`.

5.2 Performance Tracing on CloudLab

5.2.1 Monitoring Tools

For GMM modeling and ground-truth validation, we collect performance data from both switches and end hosts using Simple Network Management Protocol (SNMP), `ifconfig`, and `tcpdump`. A brief overview of each monitoring tool is provided below. For more detailed information, see the appendix (Appendix A) on monitoring tools on Cloudlab.

5.2.1.1 SNMP (Switches)

We use SNMP to collect traffic data from every network interface on all switches on Cloudlab testbed. This data includes information on incoming and outgoing bytes, packets, and discards.

By analyzing the packet and byte information obtained through SNMP, we can calculate the incoming and outgoing traffic rates (bits per second) of each interface on the core switch to check whether the links are being fully utilized or not. For example, if the measured incoming and outgoing traffic rate is close to 10 Gbps, this suggests that the corresponding link is being fully utilized.

5.2.1.2 Ifconfig (Clients & Servers)

We use the *ifconfig* command to collect packet and byte information from each server and client. This allows us to determine the number of bytes that are sent, received, and dropped between each server-client pair during our experiments. We execute the *ifconfig* command every second and measure the rate at which data are transferred between each server-client pair within each second. By analyzing this rate, we can gain insight into how bandwidth is being shared among multiple server-client pairs.

5.2.1.3 Tcpdump (Clients & Servers)

We use *tcpdump* to capture TCP/IP headers of packets transmitted and received over the network on both servers and clients during our experiments. The captured information is saved in pcap files. From these pcap files, we gather records of each HTTP request and response using 5-tuple information ($\langle src_ip, src_port, dst_ip, dst_port, protocol \rangle$) and TCP flags. Then, from each HTTP request and response, we extract performance metrics related to latency, rate, and volume. These three dimensions are relevant to the potential constraints that can limit the network performance of each transfer, as mentioned in Section 3.3. Next, we provide more details about these performance metrics below.

5.2.2 Performance Metrics

By utilizing the monitoring tools discussed above, we collect a set of performance metrics. These metrics serve two main purposes: (1) as input for building GMMs to model the network

performance of transfers, and (2) as ground-truth information to validate the modeling accuracy of the GMMs. By analyzing these performance metrics, we can identify and confirm any constraints that may be limiting the network performance of transfers in our experiments.

5.2.2.1 Delay Metrics

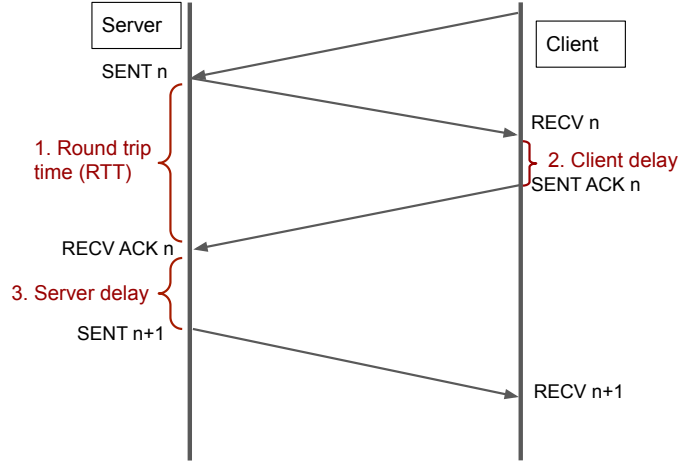


Figure 5.3: Delay information extracted using tcpdump.

We extract delay-related performance metrics on both the end hosts and the network using pcap files collected from each server-client pair. As shown in Figure 5.3, these files provide a wealth of information that can be used to determine the timing of packets and acknowledgments. On the server side, we can identify when packet N is sent out (*server.sent*), when the acknowledgment (ACK) for packet N is received (*server.recv_ack*), and when packet N+1 is sent out. On the client side, we can determine when packet N is received (*client.recv*) and when the ACK for packet N is sent out (*client.sent_ack*). Based on this information, we can calculate a set of delay-related performance metrics, as listed in Table 5.1. We do not measure end-to-end delays from servers to clients and from clients to servers over the network, as the timestamps used for calculating these metrics are collected on different machines that may not have synchronized clocks. For each HTTP request and response, we extract statistics about each type of delay and use them as input for GMMs to model their network performance in our experiments.

Types of Delays	Definition
round trip time	$server.recv_ack(n) - server.sent(n)$
client delay	$client.sent_ack(n) - client.recv(n)$
server delay	$server.sent(n+1) - server.recv_ack(n)$
network delay	round trip time - client delay

Table 5.1: A list of delay-related performance metrics extracted from pcap files.

5.2.2.2 Rate Metrics

For measuring the rate, we calculate the throughput of each transfer by dividing its response size by its transfer duration. The response size is obtained by identifying the segment with the highest TCP sequence number (tcp_seq) among all segments associated with the HTTP request-response. The response size is equal to the identified TCP segment number plus its frame length (tcp_len). The transfer duration is the time it takes for a server to transmit an HTTP response to a client. It is measured from the time the server sends the first packet of the HTTP response ($server.sent$) to the time the server receives the acknowledgement of the last packet from the client of the HTTP response ($server.recv_ack$). From the pcap trace, the $server.sent$ packet can be identified as the initial packet sent from the server to the client following the receipt of an ACK+PSH packet from the client, as the HTTP request is transmitted using TCP flags ACK+PSH. The $server.recv_ack$ packet can be identified as the packet sent by the client using TCP flags ACK+PSH+FIN. A detailed example of a pcap trace can be found in Appendix A.3. The transfer duration varies based on the size of the response. It is important to note that response sizes are only used for validating the results of GMMs, they are not used as input for building GMMs.

5.2.2.3 Volume Metrics

For describing the volume of data transmission, we extract the bytes-in-flight metric. This metrics measures the volume of data that has been sent to the network but has not yet been acknowledged by the receiving end. We extract information from pcap files to track of the number

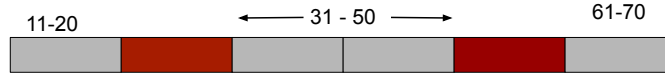


Figure 5.4: SACK visualization: cumulative ACK=21 and SACK=31-51,61-71

of bytes sent by the server and the number of bytes acknowledged by the client, and update the bytes-in-flight metric whenever the server receives an acknowledgement from the client.

TCP uses a cumulative acknowledgment scheme in which the receiver sends an acknowledgment number to the sender. This number indicates that all the bytes in the TCP stream up to the next expected byte have been received in order. In our testbed, both end hosts support TCP selective acknowledgments (SACK). With SACK enabled, the receiver can acknowledge receipt of non-consecutive data, allowing the sender to retransmit only the missing data at the receiver's end. Specifically, the receiver specifies the left and right edges of any data beyond the cumulative ACK number that has been received, in addition to the cumulative ACK. For accurately measuring bytes-in-flight, it is important to consider both cumulative and selective ACKs when tracking the total number of bytes sent by the sender and acknowledged by the receiver. In pcap files, the fields *tcp_sack_le* and *tcp_sack_re* indicate the left and right edges (the right edge is not inclusive) of segments that have already been received by the receiver, in addition to those acknowledged by cumulative ACKs. These fields can specify multiple pairs of *tcp_sack_le* and *tcp_sack_re* values.³

The bytes-in-flight is calculated by subtracting the total bytes acknowledged from the total bytes sent. This calculation is performed whenever the server sends a new segment to the client or receives an acknowledgement for a prior segment from the client. For each HTTP response, we extract statistics about the bytes-in-flight information during its transmission and use these as input features for building GMMs.

³Figure 5.4 shows an example of TCP SACK when the first (11-20), third (31-40), fourth (41-50), and sixth (61-70) packets have been received. In this case, the cumulative ACK equals 21, which is the next segment number expected by the receiver. The receiver has also selectively acknowledged two additional segments beyond the cumulative ACK number. The first segment includes the third and fourth packets (31 to 50), and the second segment includes the sixth packet (61 to 70). Thus, for the first segment, the left edge is 31 and the right edge is 51, and for the second segment, the left edge is 61 and the right edge is 71.

5.2.2.4 Cross-traffic Information

To capture the impact of cross-traffic on the network performance of network transfers, we measure the volume of cross-traffic that occurs during the lifetime of each transfer. To accomplish this, we use pcap files collected from servers and clients to determine the time interval between the request and the acknowledgement of the last segment of the response for each request-response pair. We then compute the total number of transfers and the total volume of bytes transmitted over the network from all servers during the lifetime of each transfer.⁴ This information enables us to determine the number of transfers and bytes that are competing for resources over the network and on both end hosts. Since it is not possible to obtain cross-traffic information from the trace of a single RPC, we use it only as a reference for validating the results of GMMs and not as input for building GMMs.

5.2.3 Performance Constraints on CloudLab

Table 5.2 summarizes the five types of constraints that transfers can experience on the testbed, together with the corresponding diagnosis. For instance, significant delays on either the client or server side can constrain the network performance of transfers. On the client side, delays prevent clients from sending ACKs back to servers fast enough, while on the server side, delays prevent new packets from being sent to clients quickly. Additionally, small transfers have small transmission times, so their performance primarily affected by delay. Therefore, we consider small transfers to be delay-constrained. In contrast, for large transfers with small network delay and bytes-in-flight, performance is often constrained by volume/cwnd, which prevents data from being sent out into the network quickly enough. However, when experiencing significant delays in the network, their network performance is constrained by network bandwidth.

⁴Note that this information is still subject to clock synchronization issues between different machines. However, the clock differences between machines in a cluster on Cloudlab are generally around 1-5ms (137). Given that cross-traffic information is computed throughout the duration of every RPC transfer and is primarily utilized for analyzing large transfers with an average duration of around 25ms, the impacts of clock synchronization issues are expected to be minimal.

Constraint Type	Diagnosis	Observation
Client	Clients are not sending ACKs back to servers fast enough, causing servers to slow down.	Significant local delay on clients
Server	Servers are not sending new packets to clients fast enough after receiving ACKs.	Significant local delay on servers
Delay/Size	Applications are not generating data fast enough.	Small transfer size thus small bytes-in-flight
Volume/Cwnd	Packets are not being sent out into network fast enough, either due to applications not generating data quickly enough or TCP congestion control algorithms not sending data out quickly enough.	Large transfer size but small bytes-in-flight
Network	Not enough bandwidth resources.	Significant delay over the network

Table 5.2: Types of constraints each RPC may experience in the testbed.

We conduct GMM-based analysis to understand the network performance of transfers in different scenarios by designing three sets of experiments using the above configurations and monitoring tools. These experiments evaluate the use of GMMs for: (1) revealing the network performance of transfers from the cache follower application; (2) understanding the impact of different protocol/device/traffic configurations, such as generic segmentation offloading (GSO), HTTP keep-alive, and communication patterns, on the network performance of transfers; and (3) detecting performance anomalies. Our experiments will show that GMMs are more accurate and efficient than traditional approaches for summarizing the network performance of hundreds of thousands of transfers across multiple performance dimensions, using only a small number of interpretable blobs.

5.3 Use Case 1: Understanding the Performance of Cache Application

Assessing the network performance of HDC applications and identifying any constraints that impede their performance is crucial for both application developers and data center operators. For application developers, this enables them to optimize resource configurations and enhance the efficiency and cost-effectiveness of their services. For data center operators, this facilitates the effective planning of resource upgrades and changes, enabling them to keep up with increasing demands and maintain their competitiveness.

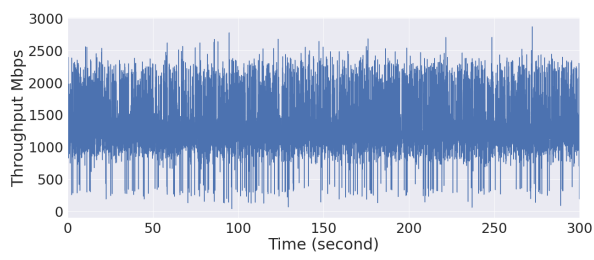
In the first experiment, we demonstrate the use of GMMs for understanding the network performance of RPCs from the cache follower application (52) in our testbed, and for identifying

bottleneck constraints that limit their performance. This experiment runs for five minutes. During this period, each client (C_n) sends multiple requests to its associated server (S_n). Each HTTP request initiates a new TCP connection. The sizes of the responses for each server-client pair follow the flow size distribution shown in Figure 5.1(a), and the time intervals between consecutive HTTP requests follow the inter-arrival time distribution shown in Figure 5.1(b). On the client side, a new request may be sent before the full response from the previous one has been received. The total number of RPCs launched in the experiment is 12,319, and each one generates a performance measurement, as outlined in Section 5.2.

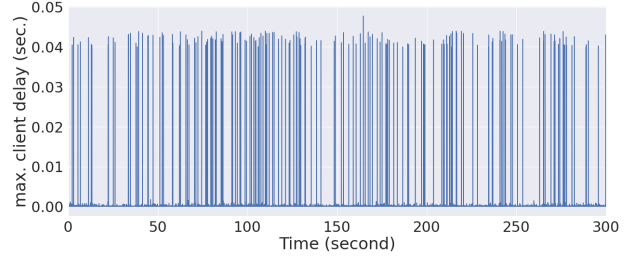
5.3.1 What Can Be Learned from Traditional Approaches

Time-series plots are a commonly utilized tool for monitoring performance in data centers. These plots enable the visualization of the evolving trends of performance-critical metrics and the detection of unusual patterns, such as spikes and outliers (9; 138; 47). For this experiment, Figure 5.5 presents time-series graphs of several performance metrics, providing an understanding of the range along each dimension. For example, the throughput can be seen to vary between 1000 to 2500 Mbps, the RTT in the 95th percentile mostly falls between 0.5 and 1 ms, and the bytes-in-flight in the 95th percentile ranges from 100 to 200 KB.

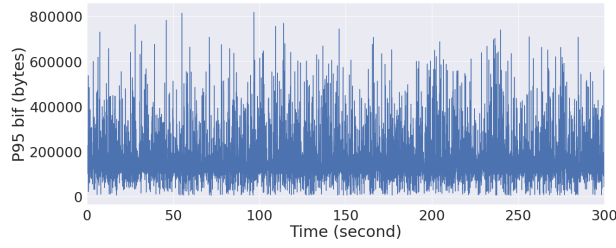
Although time-series data can be useful in illustrating the fluctuating trend of performance metrics, their spiky and wide-ranging nature can make it challenging to understand the distribution of each metric. To overcome this challenge, we can use distribution plots, such as kernel density plots (kde), to provide a clearer understanding. Kde plots uses a smooth curve to represent the distribution of data points over a continuous interval, making it easier to see the overall shape of the distribution and identify any clusters or outliers. We can also calculate statistical information about distributions, such as the median, mean, and percentile values. For example, Figure 5.6 shows kde plots of several important performance metrics for large RPCs in the experiment, along with the average values for each metric. These plots help us visualize the shape of these distributions and learn that the average throughput is approximately 1310 Mbps, the maxi-



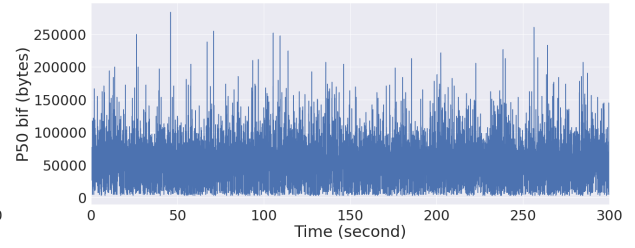
(a) throughput (mbps)



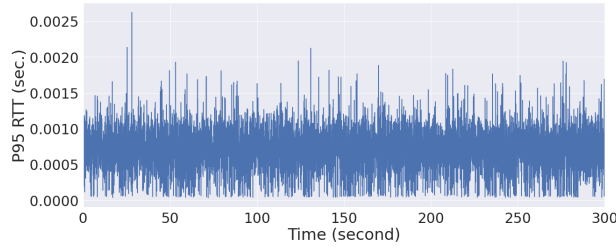
(b) max. client delay (second)



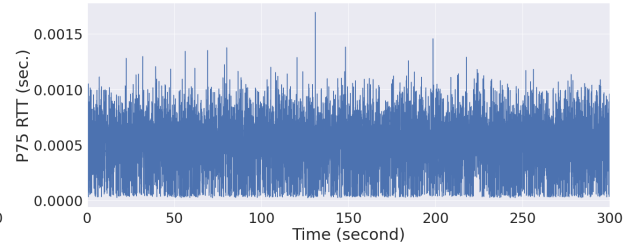
(c) 95th bytes-in-flight (MB)



(d) 50th bytes-in-flight (MB)



(e) 95th RTT (second)



(f) 75th RTT (second)

Figure 5.5: Time series of multiple performance metrics.

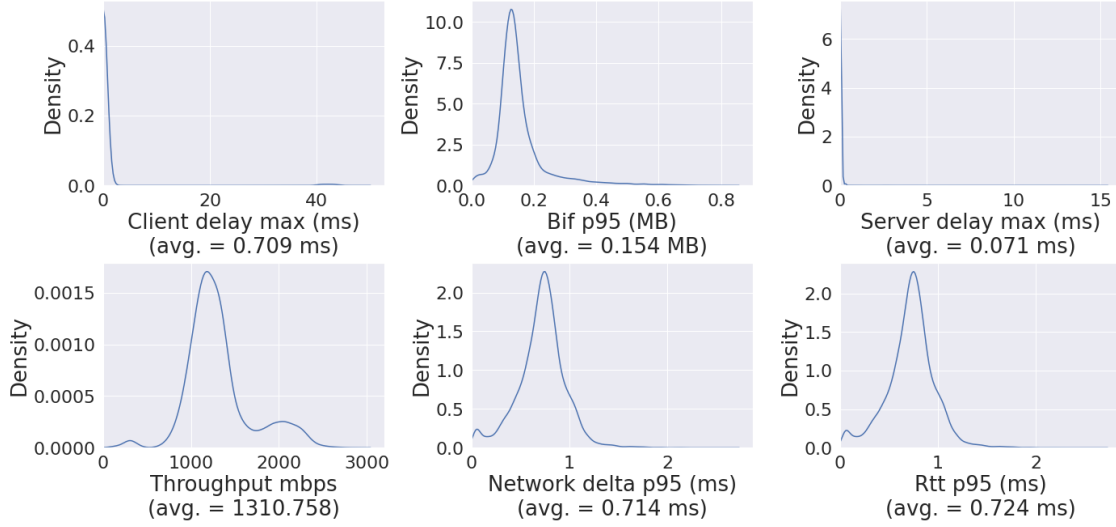


Figure 5.6: Kde plots of multiple performance dimensions in the experiment. The unit of the y-axis in a kde plot is probability density. This means that the height of the curve at a particular point on the x-axis represents the probability of a randomly selected data point falling within a very small interval around that point. It is important to note that probability density is not the same as probability. Probability density is a measure of how likely a data point is to fall within a particular range, while probability is a measure of how likely a data point is to have a particular value.

imum client delay is around 0.709 ms, and the 95th percentile RTT is 0.724 ms. However, relying on just one statistic, such as the median, mean, or 95th percentile, to characterize a distribution may not offer a complete picture of the distribution as the behavior of RPCs can vary, leading to multiple structures in the distribution. As shown in Figure 5.6, multiple modes can be observed in several performance metrics, indicating that RPCs exhibit diverse behaviors during transmission along these metrics. This is particularly evident in the throughput and 95th percentile of network delay. It is important to fully understand the distributions along each performance metric, including common behaviors represented by the mean and median, as well as tail performance (e.g., the 1st, 5th, 95th, and 99th percentiles). This is because anomalies in performance tend to occur in the tails of the distribution. For example, lower percentiles of throughput and bytes-in-flight may indicate volume-limited transfers, while higher percentiles of RTT may suggest network-limited transfers.

Furthermore, it is insufficient to analyze solely a single dimension for a comprehensive understanding of application network performance, as different dimensions are often coupled (Section

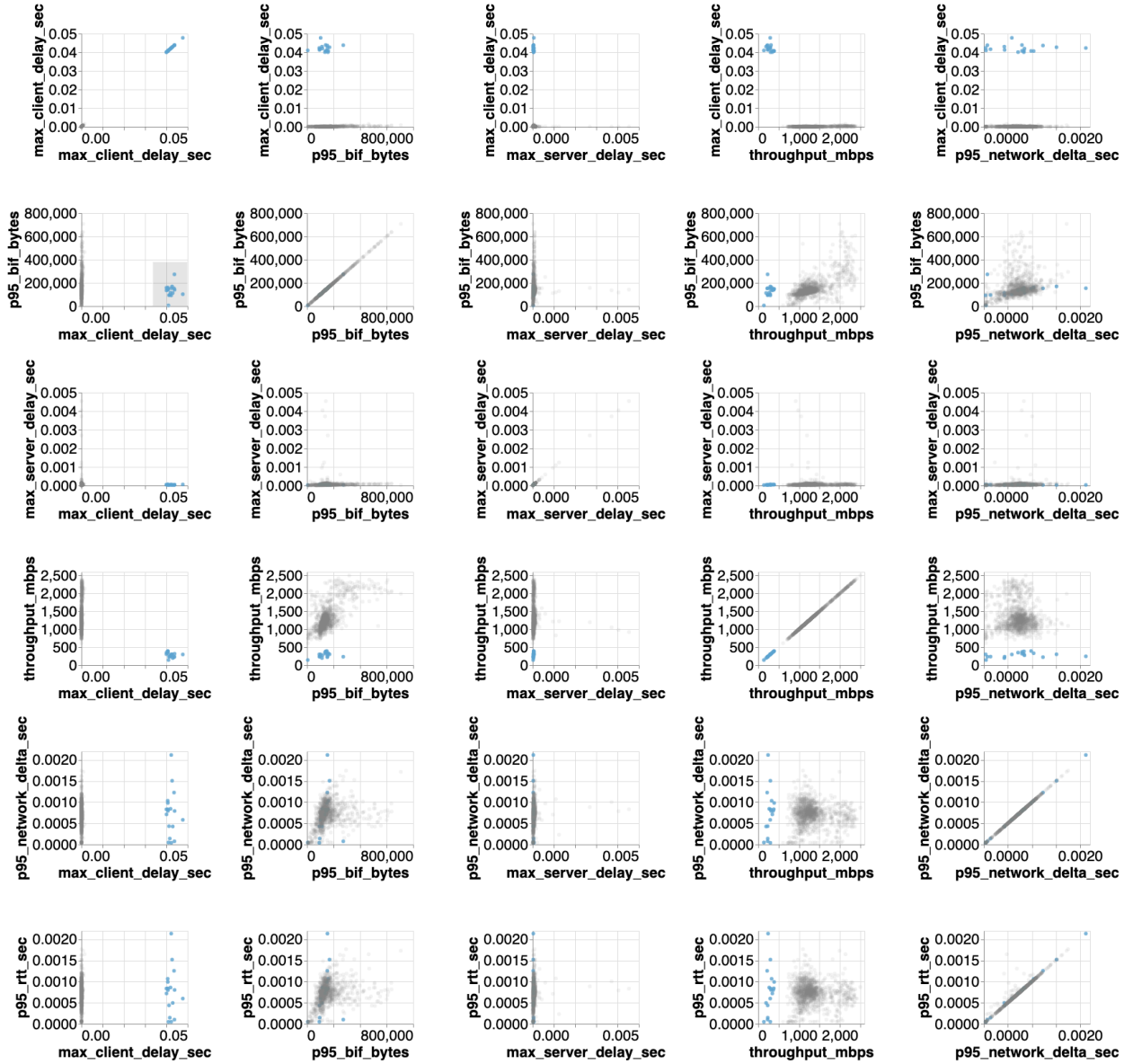


Figure 5.7: Studying the correlation of multiple performance metrics using brushing and linking with 10% of RPCs collected in the experiment.

3.3). Therefore, a multi-dimensional approach is necessary to capture these couplings and gain a complete understanding of the network performance. While traditional methods for identifying correlations between multiple performance dimensions can be time-consuming and limited, the brushing and linking approach, a common technique that allows users to interactively select and explore data by linking multiple views of the same data, can be helpful. Brushing refers to the process of selecting a subset of the data by dragging the mouse over the data of interest or using a bounding shape to isolate this subset. Linking refers to the process of connecting two or more views of the same data, such that a change to the representation in one view affects the representation in the other. As demonstrated in Figure 5.7, by selecting a subset of transfers with high *client_delay_max* and low *bif_p95* on one graph (the first graph in the second row), we can see how this subset performs in other dimensions such as throughput, server delay, and network delay — the data from the corresponding RPCs gets highlighted in the other views. We can learn that transfers with high client delay tend to have low bytes-in-flight and throughput, but their network delay is similar to other transfers. This kind of analysis can uncover patterns and trends in the data that may not be immediately noticeable by examining a single performance dimension. However, this approach becomes increasingly impractical as the volume of performance data grows. In this example, only 10% of the RPCs collected in our experiment are selected for performance metric analysis to interactively render all plots using the linked brush. In production HDCs, this method would not be feasible unless aggressive sampling is employed to reduce the size of the performance data, which could result in the loss of crucial information. Furthermore, the number of plots needed for visual analysis grows quadratically with the number of performance metrics ($O(n^2)$), making it infeasible for most scenarios.

Summary Traditional approaches for identifying patterns and trends in large-scale performance data across multiple performance dimensions are labor-intensive and error-prone. They are heavily based on manual analysis, which makes them less scalable and robust. Moreover, these approaches are not efficient in differentiating between RPCs with different performance or identifying correlations between multiple performance dimensions.

5.3.2 What Can Be Learned Using GMM Analysis

We use GMMs to model the network performance of RPCs in the experiment. The GMM modeling pipeline, outlined in Section 4.1, groups the 12,319 RPCs into 3 blobs, based on per-RPC performance metrics across multiple dimensions related to delay, rate, and volume. Out of these three blobs, blob 0 represents only 1% of the total RPCs, while blob 1 accounts for 84% of all RPCs, and blob 3 contains the remaining 15%. We then use the GMM analysis pipeline described in Section 4.2 to understand the modeling results.

The top performance metric that distinguishes the blobs is first examined using the Wasserstein distance, as listed in Table 5.3. The results reveal that blob 0 differs the most from both blobs 1 and 2 in terms of maximal client delay. Moreover, the distance between blob 0 and blobs 1(8.28)/2(8.25) is significantly larger compared to the distance of the top metric between blob 1 and blob 2 (2.14), suggesting that blob 0 exhibits more distinguishing performance characteristics, which is the local delay on the client host.

<i>Blob₁</i>	<i>Blob₂</i>	Metric	Distance
0	1	client_delay_max	8.28
0	2	client_delay_max	8.25
1	2	throughput	2.14

Table 5.3: The most distinguishing metric between GMM blobs based on the Wasserstein distance.

Figure 5.8 illustrates the breakdown of delay on local hosts and in the network for RPCs in each GMM blob. The results show that blob 0, which accounts for only about 1% of the total RPCs, is dominated by a significant local delay on the client side, while the other blobs are mainly constrained by network delay across the network. However, due to the small number of RPCs in blob 0, it can be difficult to spot the large, anomalous client delay from the overall kde plots in Figure 5.6 or to understand how these RPCs perform in other dimensions. Fortunately, GMMs provide a valuable tool for identifying and analyzing small groups of RPCs with “unique” performance characteristics in large performance data. GMMs not only separate this small group of RPCs into their own blob, but also concurrently characterize their performance in other dimen-

sions. For instance, Figures 5.8 and 5.9 reveal that, although blob 0 has server delay and network delay similar to the other blobs, it experiences much lower throughput.

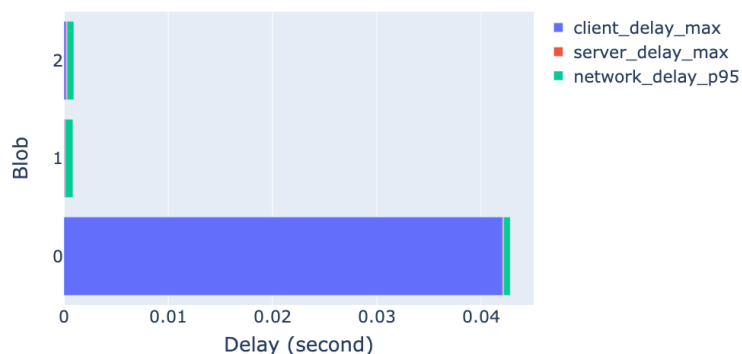


Figure 5.8: Delay information for each blob in the GMM.

Table 5.3 also highlights the difference in performance between the RPCs in blobs 1 and 2, despite their similar delays. Throughput is the most distinguishing metric, with RPCs in blob 2 having 1.7 times higher throughput and 2.5 times higher bytes-in-flight in the 95th percentile compared to blob 1. However, the size of RPCs in both blobs is similar, indicating that the lower bytes-in-flight and throughput in blob 1 are not due to size constraints, but rather volume constraints where the sender is unable to transmit data into the network at a sufficient rate. In contrast, the main constraint for RPCs in blob 2 appears to be the network, as indicated by the large bytes-in-flight and throughput.

blob ID	ratio of RPCs	constraint
0	1%	client
1	84%	volume
2	15%	network

Table 5.4: Types of constraints experienced by RPCs based on GMMs.

5.3.3 Validating Constraints with Ground Truth Information

The GMM analysis has identified the main constraints for each of the three blobs. CloudLab also enables the collection of additional performance data that are not used as input for building

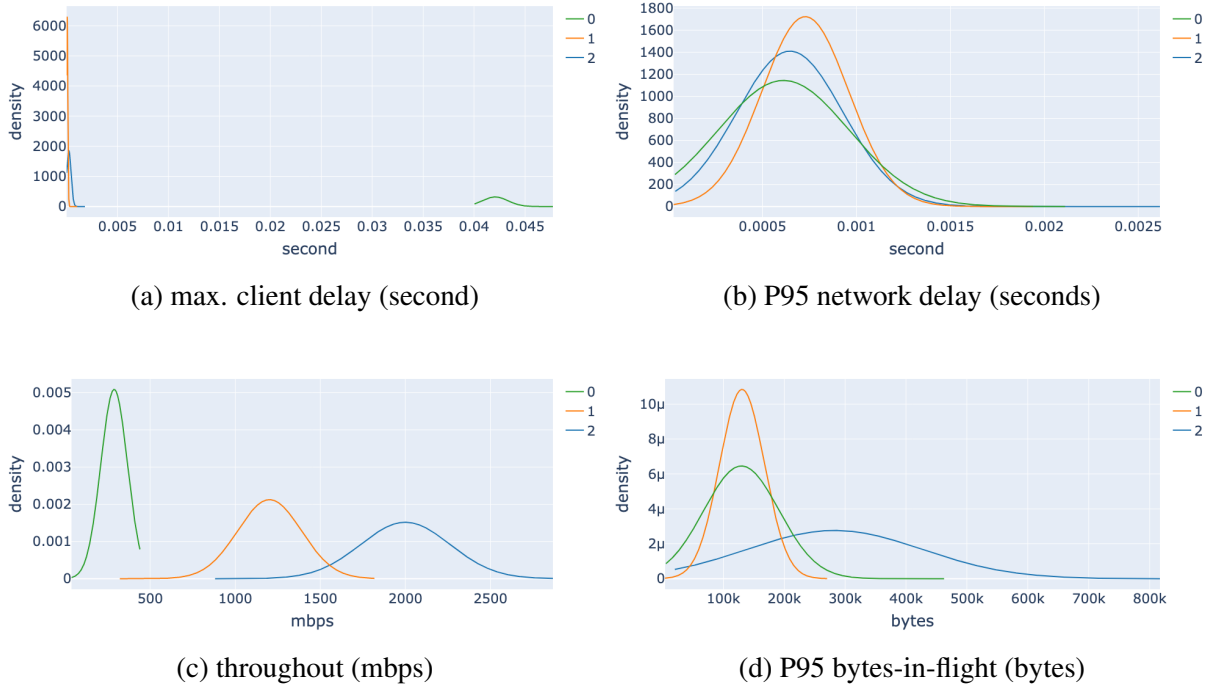


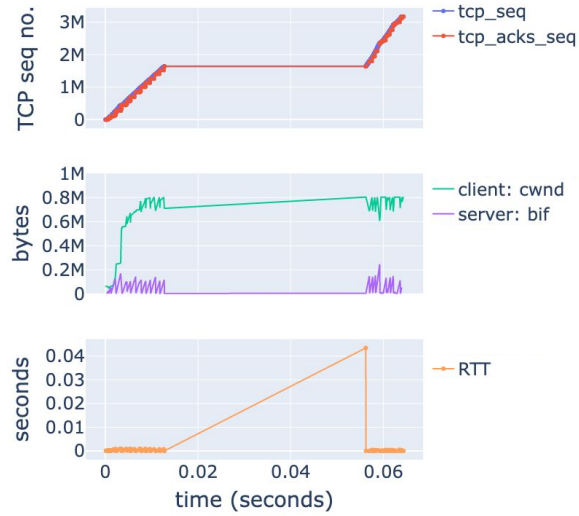
Figure 5.9: Kde plots of maximal client delay, 95th percentile of network delay, throughput, and 95th percentile of bytes-in-flight in each blob based on GMMs.

GMMs. These “ground-truth” measurements can be instrumental in validating the conclusions drawn from the GMM analysis and providing a more comprehensive understanding of the performance characteristics of the RPCs. In this section, we conduct a more in-depth analysis of the collected performance data for verifying the results obtained from the GMM analysis.

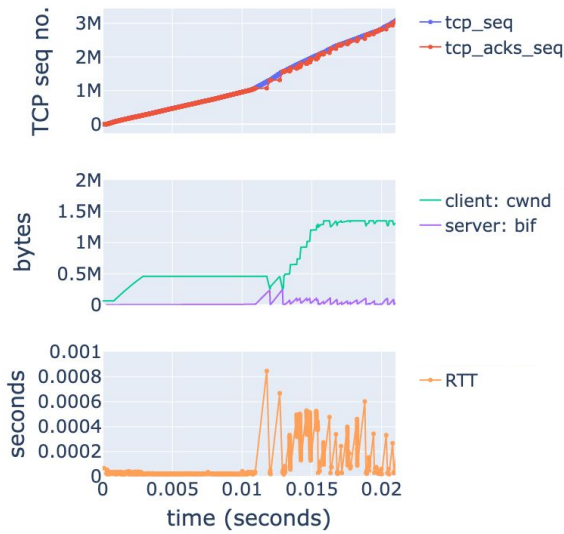
We first analyze the pcap traces from servers and clients during the experiment and find that the average number of concurrent RPCs is around 4. Given that these RPCs fairly share a 10 Gbps bottleneck link, we expect each RPC to achieve an average throughput of approximately 2500 Mbps.⁵

We then proceed to examine the time sequence graphs of the data streams for a few RPCs from each of these three blobs. The time sequence graph allows a detailed examination of every packet transmitted in each RPC between a server and a client. This includes information such as the IP address and port number, size, transmission and acknowledgement timestamps, and

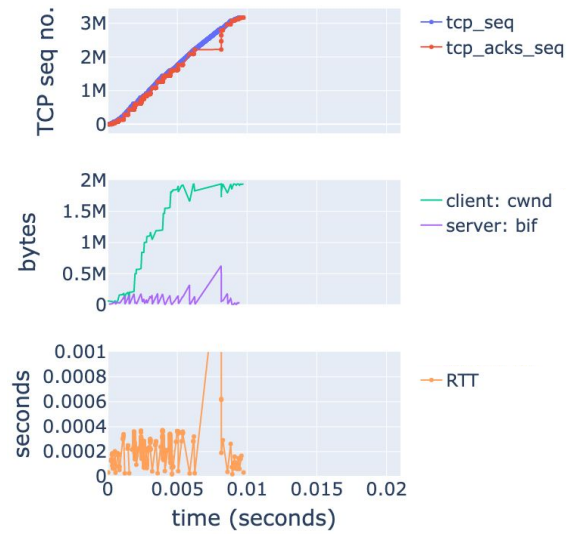
⁵It is important to note that actual throughput of each RPC may be lower than the expected value due to TCP slow-start stage and limitations in CPU processing power.



(a) blob 0



(b) blob 1



(c) blob 2

Figure 5.10: Time sequence graph of one example RPC with a size of 3,171,683 bytes in each blob based on pcap files captured on the server end.

the size of the congestion window (as described in Appendix A). This information is not used as input for building GMMs, GMMs instead use performance statistics and derivatives extracted from each RPC, such as the 95th percentile of RTT and the delivery rate. Therefore, compared to the information used for GMM modeling, the time sequence graphs offer a more detailed and comprehensive analysis of the behavior of each RPC.

Our analysis finds that the time-sequence graphs for RPCs within a particular blob are relatively similar, but differ significantly for RPCs in different blobs. We present a sample trace of an RPC from each GMM blob to demonstrate the temporal progression (shown in Figure 5.10). To ensure a fair comparison, all sample RPCs have the same size. For each sample RPC, we present time sequence plots of the following: (1) TCP sequence number (*tcp_seq*) and TCP acknowledgment sequence number (*tcp_ack_seq*) to show the number of bytes sent and acknowledged; (2) TCP receiver window size advertised by the client (*client: cwnd*) to indicate the amount of data that the server can send without overflowing the client, and the number of bytes-in-flight on the server side (*server: bytes-in-flight*) to represent the number of bytes sent but not yet acknowledged; (3) round-trip time (*RTT*), which is the time between the packet being sent and the corresponding acknowledgment being received by the server. Note that RTTs in this case also include local delay on the client side (as shown in Figure 5.3). We next examine the time sequence graphs.

As shown by *tcp_seq* and *tcp_ack_seq* plots in Figure 5.10(a), the sample RPC in blob 0 experiences a pause of approximately 44 milliseconds (nearly 68% of the total transfer time) while waiting for an acknowledgment. Once the acknowledgement is received, the transfer resumes without further disruptions. During this pause, the measured RTT increases suddenly, from less than 0.001 seconds to 0.043 seconds, as a result of the delayed arrival of the acknowledgment. For determining whether the increase in RTT occurred in the network or on the client side, we analyze the local delay recorded on the client side, as shown in Figure 5.11. The local delay rises from around 6 milliseconds to 43 milliseconds, contributing to the increase in RTT. The pcap traces support the conclusion that the performance of the RPCs in blob 0 is mainly impacted

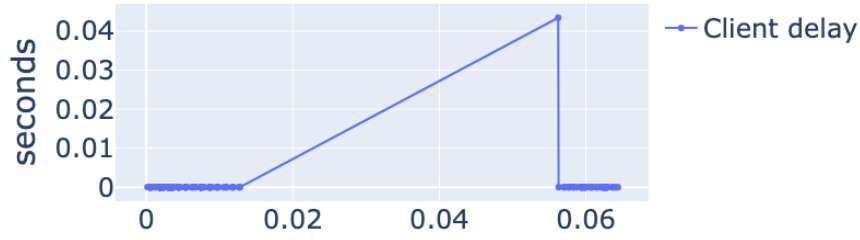


Figure 5.11: Time sequence graph of client delay for the example RPC in Figure 5.10(a) based on pcap files captured on the client end.

by an increase in local delay on the client. Therefore, blob 0 is mainly constrained by clients. Categorical analysis shows that blob 0 contains RPCs from all clients, not just from a particular one(s). Therefore, the performance bottleneck is not due to anomalous clients, but rather a more generic performance bottleneck RPCs may experience during transmission.

The time sequence graphs for the sample RPCs in blobs 1 and 2 are shown in Figures 5.10(b) and 5.10(c), respectively. Unlike the RPC in blob 0, as shown in Figure 5.10(a), these RPCs do not experience significant local delays on the client side. However, the RPC in blob 1 has a much smaller cwnd and fewer bytes-in-flight at the start of the transfer compared to the RPC in blob 2. This results in a 2 times longer time to transmit the same amount of data, given that the size of these RPCs is the same. For instance, the number of bytes-in-flight remains around 4 KB and the cwnd is around 458 KB during the first 0.01 seconds for the RPC in blob 1. Meanwhile, the RTTs stabilize around 0.05 milliseconds, which is closer to the propagation delay.⁶ The small RTTs suggest that the network is not the constraint for RPCs in blob 1. Instead, the performance is primarily constrained by volume/cwnd, where the sender is unable to generate/transmit data quickly enough, resulting in underutilization of the available bandwidth resources within the network.

⁶The minimal RTT measured using the *ping* command between a server and a client in our testbed is around 0.05 milliseconds.

In contrast, the RPC in blob 2 completes in less than 0.01 seconds, with an average of 136 KB in bytes-in-flight and a rapidly increasing cwnd, as shown in Figure 5.10(c). The average throughput of RPCs in blob 2 is around 2000 Mbps (Figure 5.9(c)), which is close to the expected throughput of 2500 Mbps when 4 RPCs, the average number of concurrent RPCs during the experiment, fairly share a 10 Gbps bottleneck link. This indicates that the RPCs in blob 2 effectively utilize bandwidth resources and their performance is mainly limited by the network.

In conclusion, our analysis reveals that the performance of the RPCs in each GMM blob is influenced by different factors. The RPCs in blob 0 are mainly constrained by clients with high local latency, those in blob 1 by volume with small bytes-in-flight during transfer, and those in blob 2 by network with high throughput and bytes-in-flight. This demonstrates that the physically based GMM model can effectively group RPCs based on their network performance, even without the use of detailed per-packet time sequence information.

Unlike traditional approaches, GMMs offer a concise summary of tens of thousands of RPCs through three multidimensional Gaussian distributions (blobs). This simplifies the process of: (1) Identifying abnormal performance experienced by a small group of RPCs (blob 0). (2) Summarizing the network performance of RPCs across multiple dimensions. (3) Interpret the main constraint associated with each blob.

5.3.4 GMMs vs. kMeans vs. DBSCAN

As explained in Section 3.2.2, our use of GMM is based on the physical interpretation of a large number of similarly-constrained application workers in HDCs, rather than simply clustering based on performance metrics. To demonstrate this, we compare the modeling results of GMMs with two commonly used clustering approaches, kMeans (139) and DBSCAN (140), to determine if they can also identify the three blobs and their unique constraints. For a fair comparison, we use the same set of RPCs and the same set of performance metrics as input for all modeling approaches.

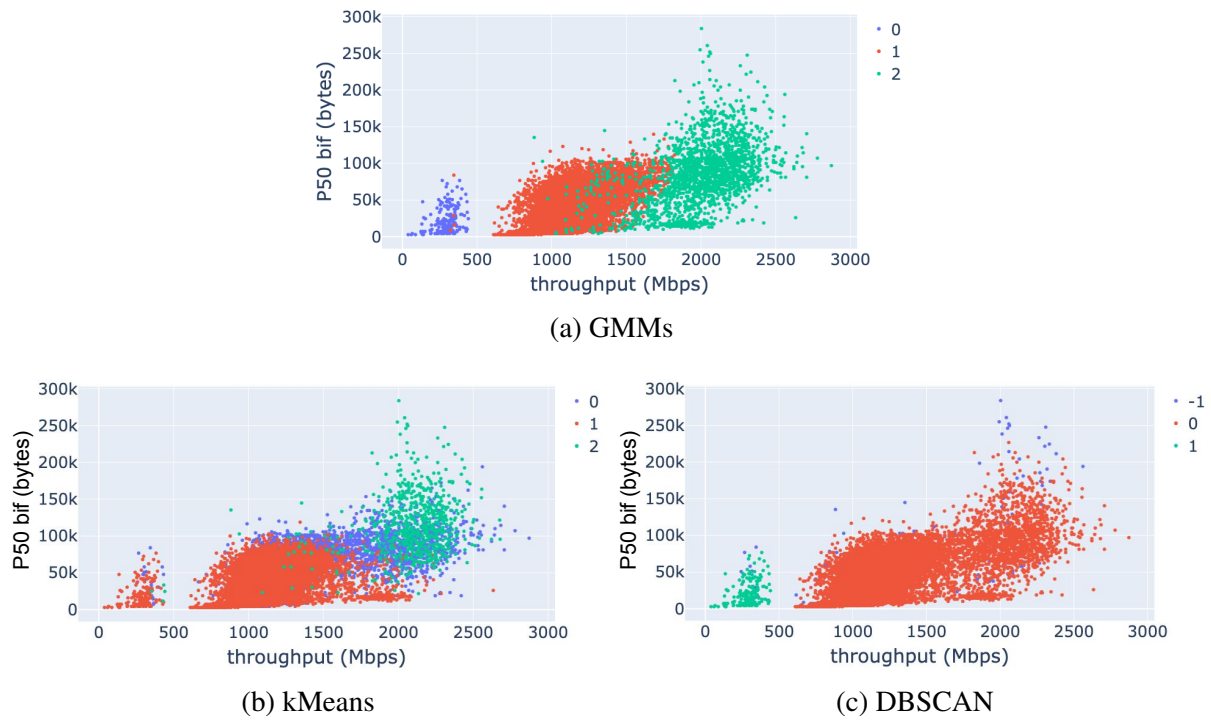


Figure 5.12: Scatter plots in throughput and 50th bytes-in-flight with different clustering algorithms. Note that DBSCAN uses the value -1 to indicate outliers that do not belong to any cluster. This suggests that DBSCAN is not effective in this case, as it is unable to cluster all of the data points.

The scatter plots in Figure 5.12 compare the throughput and 50th percentile of bytes-in-flight for each blob using three different clustering algorithms. We compare the RPCs clustered in each blob using different approaches. The results indicate that kMeans cannot identify client-constrained RPCs, as these only constitutes 1% of the performance data. On the other hand, DBSCAN can identify client-constrained RPCs, but fails to differentiate between volume-constrained and network-constrained RPCs due to overlap in their performance near the edges.

The results of the comparison between GMMs, kMeans, and DBSCAN show that GMMs provide a more accurate identification of different network performance experienced by RPCs and their constraints. GMM-based analysis is able to handle disproportionate ratios of RPCs across blobs and similar performance along one or more individual dimensions, which are limitations of kMeans and DBSCAN.

5.4 Use Case 2: Analyzing the Impact of Configuration Factors on RPC Performance

In this section, we employ GMMs to investigate the impact of configuration factors such as generic segmentation offloading (GSO) (Section 5.4.1), HTTP Keep-Alive (Section 5.4.2 and Appendix B.1), and communication patterns (Section 5.4.3 and Appendix B.2) on the network performance of RPCs by conducting A/B comparisons.

5.4.1 Experiment 1: The Impact of Generic Segmentation Offloading (GSO)

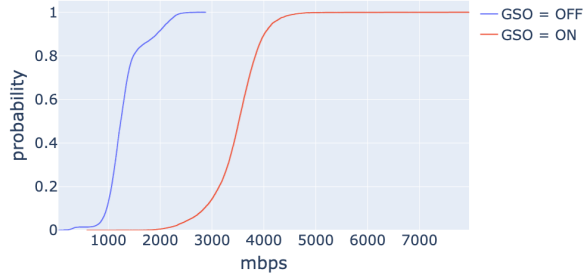
In the previous experiment (Section 5.3), it was observed that 84% of the RPCs are limited by volume, as indicated by their low throughput and bytes-in-flight (blob 1 in Table 5.4). We had disabled GSO during the experiment, and wondered if that resulted in poor performance. To investigate the effect of GSO on the network performance of RPCs, we repeat the previous experiment with GSO enabled by executing the *ethtool* command, using the same setup as described in Section 5.3.

5.4.1.1 Background: GSO

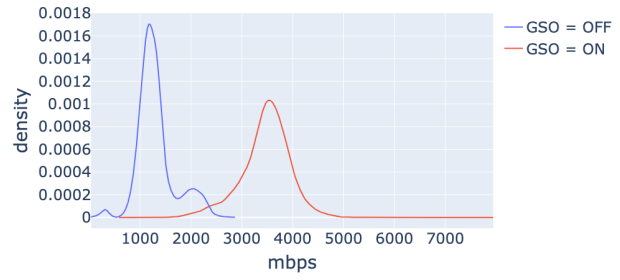
GSO is a technique used in networking to optimize the transmission of large data packets over networks with a limited Maximum Transmission Unit (MTU). The MTU refers to the maximum size of a packet that can be transmitted over a network without fragmentation. When a network interface receives a large packet that exceeds the MTU, it needs to be fragmented into smaller packets before transmission. This fragmentation process can introduce overhead and inefficiencies, especially when dealing with large-scale data transfers. GSO addresses this issue by offloading the segmentation process from the sending device's CPU to the network interface hardware, thus enables upper-layer applications to handle fewer, larger packets instead of more, smaller packets, and reduces the per-packet overhead in the network stack. As a result, GSO can lead to more efficient packet sending and receiving when it is enabled on the end hosts.

5.4.1.2 What Can Be Learned from Traditional Analysis

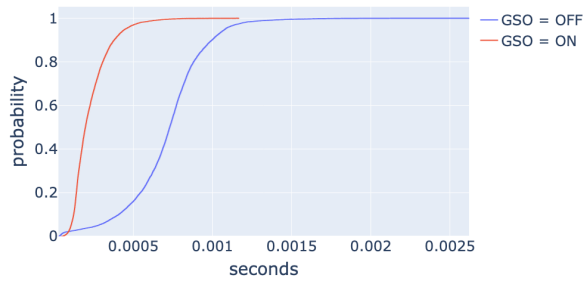
The comparison of network delay, throughput, and bytes-in-flight with and without GSO is shown in Figure 5.13 using cdf and kde plots. The results show that when GSO is enabled, the average network delay in the 95th percentile decreases by approximately 70%, from 0.7 to 0.2 milliseconds. Additionally, the average throughput increases by 2.7 times, from 1315 to 3472 Mbps, and the average bytes-in-flight in the 95th percentile increases by 1.48 times, from 154 to 228 KB. These positive results align with the design goal of GSO. However, the kde plots in Figure 5.13 also reveal multiple structures in throughput and bytes-in-flight when GSO is enabled, suggesting that different RPCs experience diverse performance. Investigating the correlations between these different modes across performance dimensions is a complex task that can be accomplished using techniques similar to the linking and brushing method shown in Figure 5.7.



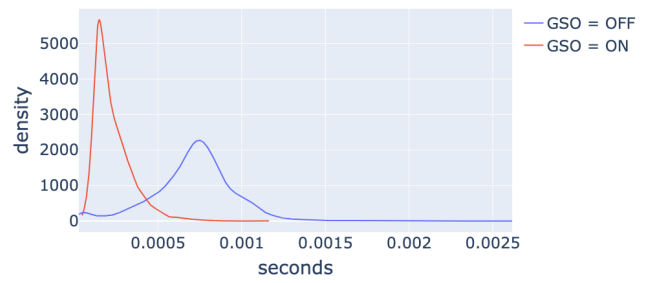
(a) Cumulative distribution of throughput



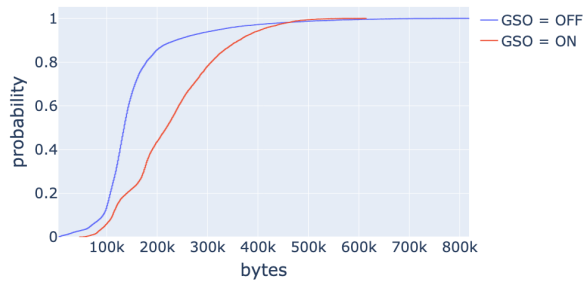
(b) Kde of throughput



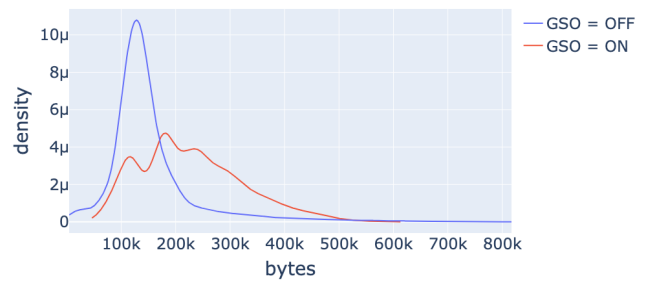
(c) Cumulative distribution of P95 network delay



(d) Kde of P95 network delay



(e) Cumulative distribution of P95 bytes-in-flight



(f) Kde of P95 bytes-in-flight

Figure 5.13: Performance comparison in the 95th percentile network delay, throughput and 95th percentile bytes-in-flight with and without GSO.

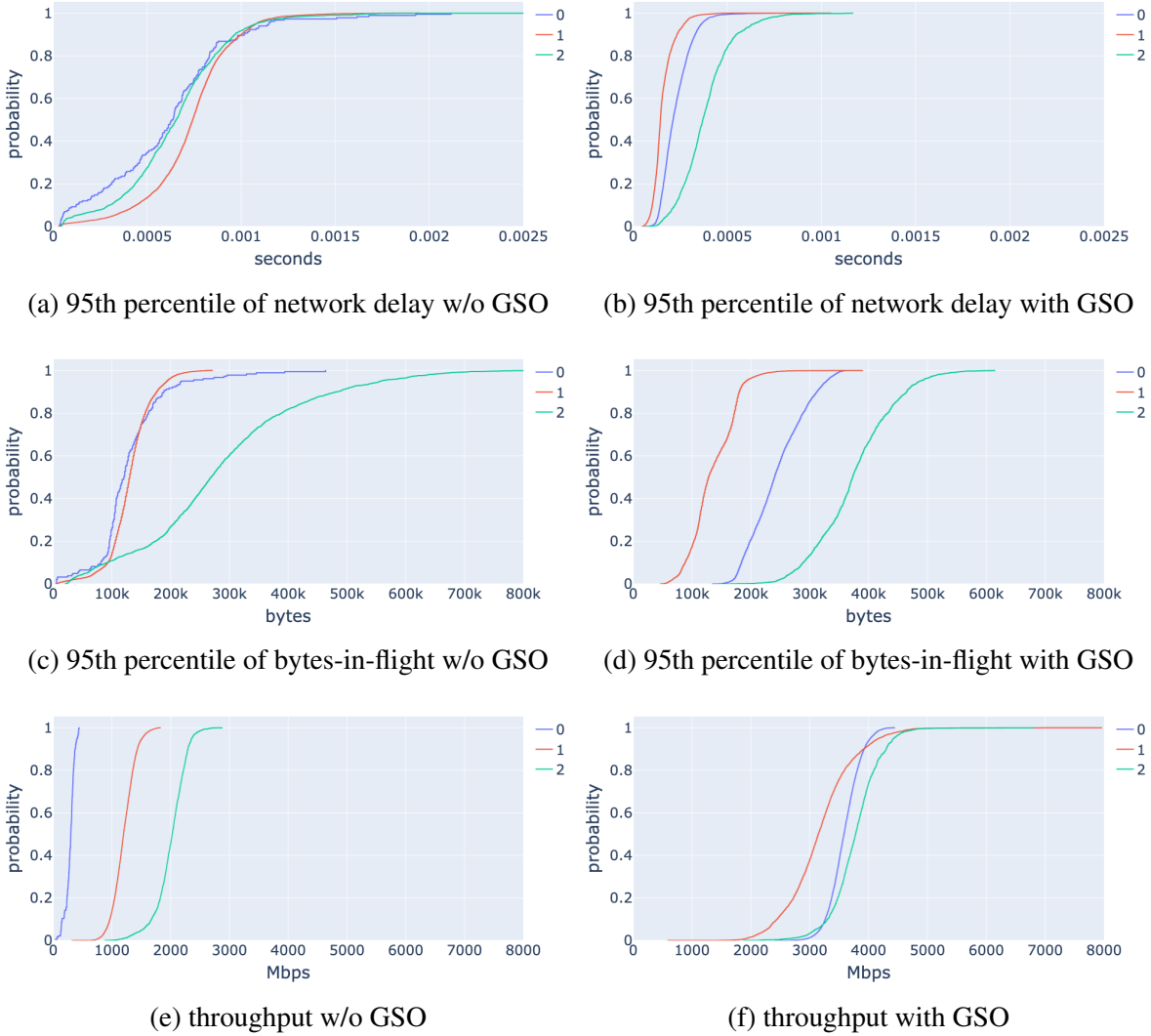


Figure 5.14: Cumulative probability of multiple performance metrics of each blob with and without GSO.

5.4.1.3 What Can Be Learned Using GMM Analysis

To avoid the overhead of traditional linking and brushing techniques, we construct GMMs using the same set of per-RPC performance metrics utilized as in the previous experiment when GSO is disabled (Section 5.3). We then apply the blob matching methodology outlined in Section 4.2.3 to match the blobs in these two GMMs, with and without GSO, for an “apples-to-apples” comparison. Matched blobs are assigned the same label for annotation. By comparing the GMM results of the matched blobs, we aim to uncover any supplementary insights that may emerge.

Blob (RPC ratio)	Network delay	Throughout (Mbps)	P95 bif (Kbytes)
0 (1%)	0.61	288.4	129.73
1 (84%)	0.73	1201.4	130.71
2 (15%)	0.65	2000.1	282.76

a GSO = OFF

Blob (RPC ratio)	Network delay	Throughout (Mbps)	P95 bif (Kbytes)
0 (57%)	0.22	3585.4	244.35
1 (23%)	0.15	3169.4	135.30
2 (20%)	0.38	3775.1	375.25

b GSO = ON

Table 5.5: Average performance of RPCs in each blob.

Figure 5.14 presents the cumulative distribution of the most distinguishing metrics between each blob according to the Wasserstein distance: 95th percentile of network delay, 95th percentile of bytes-in-flight, and throughput. Their average performance is summarized in Table 5.5. As seen in the table, when GSO is disabled, the volume-constrained blob 1 and network-constrained blob 2 show a 160% difference in throughput (1201 Mbps vs. 2000 Mbps) and a 210% difference in the 95th percentile of bytes-in-flights (129 KB vs. 282 KB). However, when GSO is enabled, the largest gap between blobs in the GMM is reduced to 1.2x difference in throughput (3169 Mbps vs. 3775 Mbps) and a 1.5x difference in the 95th percentile bytes-in-flight (244 KB vs. 375 KB), resulting in more consistent performance for all RPCs. This demonstrates that with GSO enabled, there is a more uniform distribution of performance across the blobs, improving fairness.

As shown in Figure 5.14, the performance characteristics of three blobs in the GMM with GSO enabled reveal that the RPCs in blob 2 experience the highest network delay and throughput. This suggests the presence of additional queuing delay introduced in the network during transmissions. Queuing occurs when the bottleneck link is saturated and the bandwidth is sufficiently utilized, indicating that the performance of RPCs in blob 2 is mainly constrained by the network. Since there is no significant delay at either end for all blobs as depicted in Figure 5.15, the constraint for RPCs in blobs 0 and 1 is either size or volume given their small bytes-in-flight. However, since the RPCs in blob 0 experience a similar throughput as the RPCs in the network-constrained blob 2, it is unlikely that their performance is limited by size.



Figure 5.15: Delay breakdown in each blob with and without GSO.

Additionally, Figure 5.15 shows the breakdown of the most distinguishing delay metrics in the network and on the end hosts for each blob in both GMMs. One of the most notable observations from the plot is that the significant delay observed on the client side when GSO is disabled is no longer present when GSO is enabled, suggesting that the RPCs are no longer constrained by the client in this scenario. Because the large client delay experienced by the small set of RPCs is not clearly visible in the kde plot when GSO is disabled (Figure 5.6), it is challenging to highlight this change based only on visualization. Additionally, the network delay experienced by the RPCs in each blob is lower, which aligns with the overall trend⁷.

In conclusion, the GMM analysis reveals that the possible constraint for blob 0 is volume, for blob 1 is size, and for blob 2 is the network. Moreover, as the experiment demonstrates, changing a single factor (i.e., enabling or disabling GSO) can lead to new interactions among the RPC flows and with the network, often in unpredictable ways. In this case, GMM is very useful in illustrating the different influences on subsets of RPC flows.

⁷The reduction in network delay may seem counterintuitive since GSO is not expected to impact network delay. This unexpected result can be attributed to the imprecise timestamps captured in tcpdump. These timestamps represent the approximate arrival time of incoming packets and the transmission time of outgoing packets, but they may not be entirely accurate due to factors such as delayed processing caused by system interrupts. In particular, when GSO is disabled, the arrival timestamp for a packet may be delayed if the system is busy handling smaller packets, even though this delay is not introduced over the network. As a result, the delay may be erroneously counted as part of the network delay in the timestamps captured in tcpdump.

5.4.1.4 Validating Constraints Using Ground Truth Information

Next, we validate the constraints associated with each blob in the GMM with GSO enabled by using ground-truth information from the pcap traces. The cumulative distribution of the size of each RPC for each blob is shown in Figure 5.16. As the figure demonstrates, the RPCs in blob 1 have a smaller median size, approximately 800 KB, compared to those in blobs 0 and 2, which have a median size of around 2.4 MB. This smaller size may contribute to their performance characteristics, such as lower throughput and fewer bytes-in-flight, as smaller RPCs may spend a larger proportion of time in the TCP slow-start stage. Therefore, the network performance of RPCs in blob 1 is mainly limited by their sizes. It is noteworthy that the size of RPCs is not included as an input performance metric when modeling GMMs, as its distribution may not be Gaussian and can vary depending on the specific application. However, GMMs can still identify the impact of RPC size on network performance by analyzing other performance metrics.

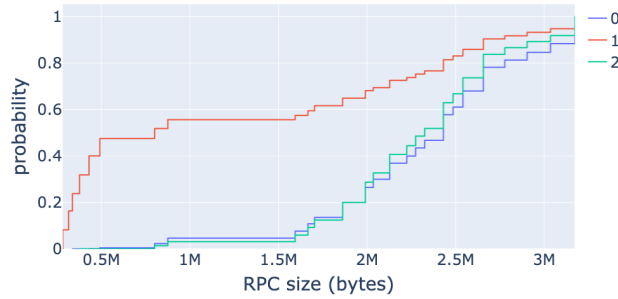
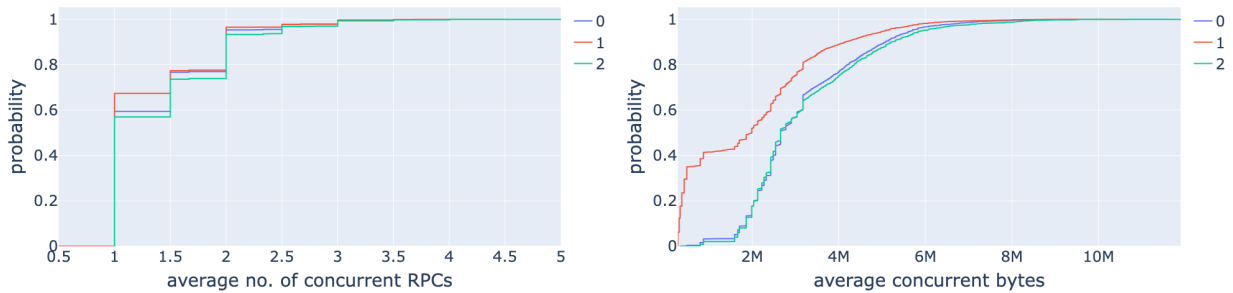


Figure 5.16: Cumulative distribution of RPC sizes of each GMM blob when GSO is on.



(a) Cumulative distribution of RPC numbers.

(b) Cumulative distribution of RPC bytes.

Figure 5.17: Cross-traffic information of each GMM blob when GSO is on.

To gain insights into the performance differences between the RPCs in blobs 0 and 2, we conduct a further analysis of the pcap files. We evaluate the cross-traffic to assess the network load during the experiment. By combining information from all pcap files, we calculate the average number of concurrent RPCs and their bytes at any given moment. As shown in Figure 5.17, it is evident that, during approximately 80% of the observed duration, the number of concurrent RPCs within the network does not exceed 2. Moreover, when examining the cross-traffic bytes associated with RPCs in blobs 0 and 2, they exhibit similarity, indicating the absence of any notable differences. Thus, the performance differences observed between blobs 0 and 2 cannot be attributed to cross-traffic influences.

Examination of time sequence graphs of sample RPCs from blobs 0 and 2, with similar sizes, reveals some key insights. As shown in Figure 5.18, the RPC in blob 2 completes 1 millisecond faster, with higher bytes-in-flight and smaller RTTs during transmission, compared to the RPC in blob 0. The increasing RTTs observed in the graph of the RPC in blob 2 (Figure 5.18(b)) as bytes-in-flight increases suggest that additional queueing delays are introduced and the network bandwidth is limited. This confirms that the network is the primary constraint for the performance of the RPCs in Blob 2. In contrast, the RPCs in blob 0 exhibit smaller RTTs and lower throughput, suggesting that their performance may be more constrained by volume, rather than network conditions. This indicates that the RPCs in blob 0 do not utilize the available bandwidth as efficiently as those in blob 2.

The conclusion drawn from the pcap files regarding the performance constraints for each blob is consistent with the analysis based on GMM results.

5.4.2 Experiment 2: The Impact of HTTP Persistence

In this section, we investigate the ability of GMMs to assess the effect of the persistence of HTTP connections on the network performance of RPCs. HTTP Keep-Alive, also known as an HTTP persistent connection, enables a single TCP connection to remain open for multiple HTTP requests and responses, instead of creating a new connection for each request and response. This

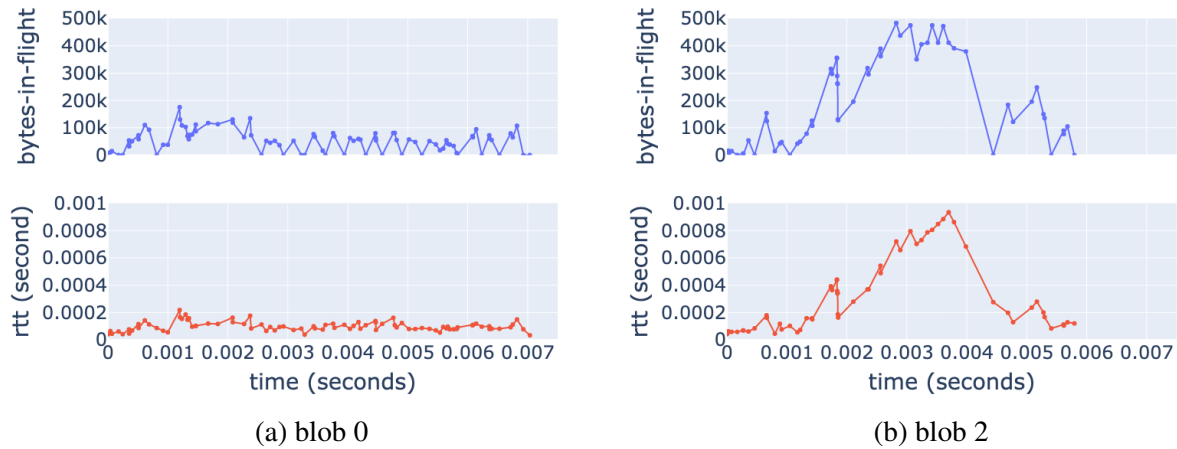


Figure 5.18: Time sequences of bytes-in-flight and RTT for sample traces in blobs 0 and 2.

reduces the overhead of connection establishment and termination, leading to faster data transfer, as it avoids the overhead of establishing and tearing down TCP connections for each request.⁸

Figure 5.19 illustrates the difference in TCP time sequence graphs with and without HTTP Keep-Alive. When a new connection is established without HTTP Keep-Alive, the bytes-in-flights on the server side (server:bif), which is limited by the congestion window size (cwnd), starts with a small value and gradually increases, as shown in Figure 5.19(a). However, when Keep-Alive is enabled, the connection continues with the large cwnd (and possible bytes-in-flight) from the previous requests, as depicted in Figure 5.19(b). This leads to improved performance with a shorter transfer duration and larger throughput. Given its performance advantages, HTTP Keep-Alive is widely used in data centers to maintain open connections between machines. Therefore, it is crucial to understand its impact on the network performance of RPCs in our testbed using traffic patterns similar to those found in data centers. To evaluate the impact of HTTP Keep-Alive on RPC performance, we conduct experiments under two different workloads: (1) with at most one connection between each server/client pair (Appendix B.1), and (2) with up to five connections between each server/client pair.

⁸However, there are certain cases in which HTTP Keep-Alive cannot bypass TCP slow start. For instance, when the TCP connection between the client and server exceeds its idle timeout period, either party may close the connection. In such cases, when the client initiates a subsequent request, a new TCP connection is established, triggering TCP slow start once more.

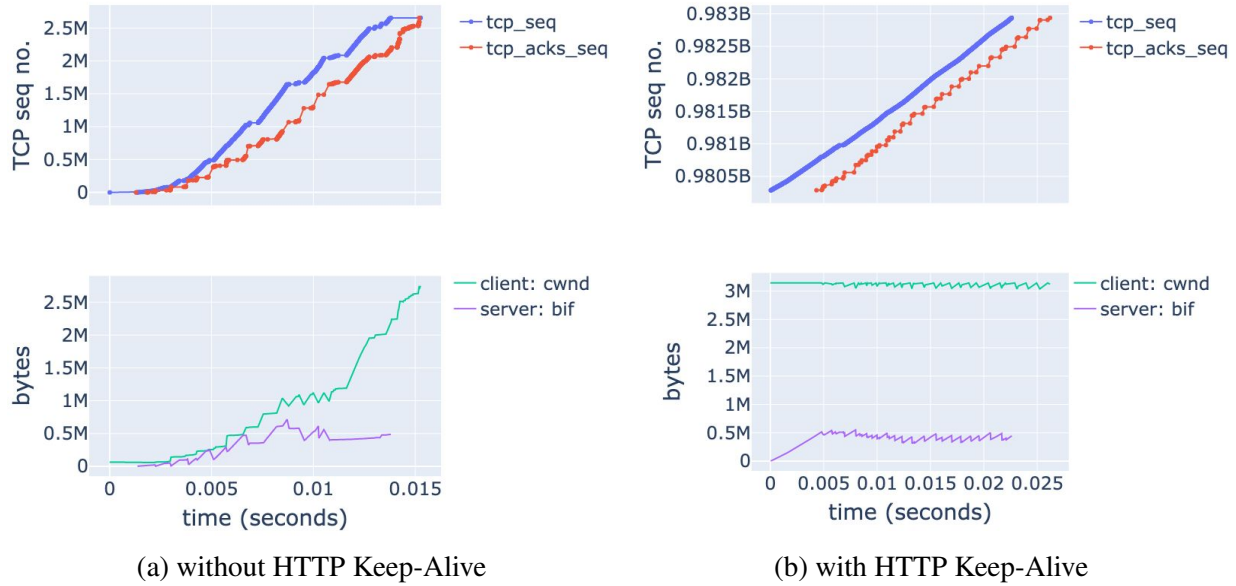


Figure 5.19: TCP time-sequence graph without (left) and with (right) HTTP Keep-Alive. Note that the time in (a) corresponds to a single request, while the time in (b) is for a later request sent after the establishment of a keep-alive connection.

5.4.2.1 The Impact of HTTP Keep-Alive with Multiple Connections

In order to study the effect of HTTP Keep-Alive on the network performance of RPCs when there is increased workload, we establish five connections between each server/client pair. Similar to the single-connection scenario, each connection transferred 1 GB of data through multiple HTTP requests and responses, resulting in a total data transfer of 5 GB between each server-client pair. The distribution of response sizes remains consistent with the prior experiments, and each client stops issuing new requests to a server after receiving responses totaling 1 GB in size. This test scenario is relevant as HTTP Keep-Alive is commonly used in data centers, and it enables us to assess its impact on RPC performance under higher loads.

What Can Be Learned from Traditional Analysis Table 5.6 compares the average performance of RPCs with and without the use of HTTP Keep-Alive with multiple connections. As the results indicate, enabling HTTP Keep-Alive leads to a decrease in the overall completion rate from 8.4 Gbps to 8.27 Gbps. Both values are close to the bottleneck link speed of 10 Gbps, indi-

cating high utilization of the network. In comparison, the single connection scenario, as shown in Table B.1, exhibits a completion rate of around 4.78 Gbps. Additionally, enabling HTTP Keep-Alive leads to a decrease in average throughput from 536 Mbps to 341 Mbps, a reduction in the 95th percentile of bytes-in-flight from 0.57 MB to 0.49 MB, and an increase in the 95th percentile of RTT from 4.58 ms to 10.88 ms. These results suggest that when HTTP Keep-Alive is enabled under increased workload, the network performance of RPCs is degraded.

HTTP Keep-Alive	Overall rate	Throughput	95th Bif	95th Rtt
enabled	8.27 <i>Gbps</i>	341 <i>Mbps</i>	0.49 <i>MB</i>	10.88 <i>ms</i>
disabled	8.40 <i>Gbps</i>	536 <i>Mbps</i>	0.57 <i>MB</i>	4.58 <i>ms</i>

Table 5.6: Comparison of average performance with and without HTTP Keep-Alive for multiple connections.

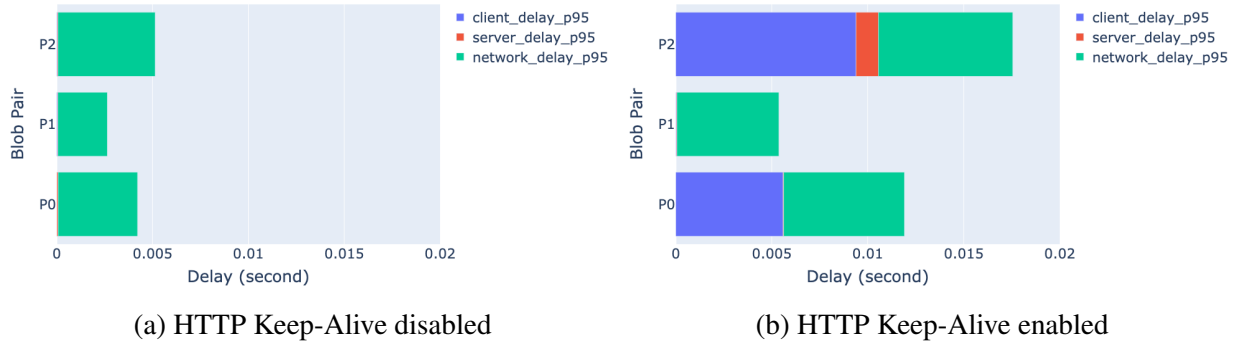


Figure 5.20: Breakdown of delay on end-hosts and network for each blob in GMMs with multiple connections.

	$P0_{disable}$	$P1_{disable}$	$P2_{disable}$
$P0_{enable}$	12.84	17.52	21.88
$P1_{enable}$	24.86	10.98	18.21
$P2_{enable}$	13.85	14.85	3.53

Table 5.7: Distance measure of the 2-dimensional Wasserstein distance between blobs in two GMMs.

What Can Be Learned Using GMM Analysis The previous summary provides an overview of the performance of RPCs with and without HTTP Keep-Alive, but does not offer a comprehensive understanding of how HTTP Keep-Alive impacts RPCs. To gain deeper insight, we employ

GMMs to build two models based on per-RPC metrics collected with and without HTTP Keep-Alive. The resulting blobs in these models are then matched using 2d Wasserstein distance, as described in Section 4.2.3, to facilitate an “apples-to-apples” comparison. This matching process leads to the identification of three pairs of matched blobs, which are denoted as $P0_{disable}/P0_{enable}$, $P1_{disable}/P1_{enable}$, and $P2_{disable}/P2_{enable}$, respectively.

Figure 5.20 presents a comparison of the breakdown of delay on end hosts and in the network for each blob in these GMMs in the case of multiple connections, with and without HTTP Keep-Alive. When HTTP Keep-Alive is disabled (Figure 5.20(a)), the network performance of RPCs is predominantly limited by the network delay, which is the most significant factor. In contrast, when HTTP Keep-Alive is enabled (Figure 5.20(b)), there is a noticeable increase in local client delay, along with an increase in network delay. Table 5.8 summarizes the top 5 performance metrics that differentiate these blobs in each GMM. When HTTP Keep-Alive is disabled, the performance of RPCs in the different blobs varies in terms of throughput, server delay, bytes-in-flight, network delay, and RTT. On the other hand, when HTTP Keep-Alive is enabled, the performance of RPCs mainly differs in terms of RTTs and bytes-in-flight.

	without HTTP Keep-Alive		with HTTP Keep-Alive	
	performance metric	distance	performance metric	distance
1	throughput	1.69	rtt_p50	0.79
2	server_delay_p75	1.36	bif_p50	0.73
3	bif_p95	1.27	rtt_mean	0.73
4	network_delay_mean	1.26	bif_mean	0.69
5	rtt_mean	1.26	bif_p75	0.69

Table 5.8: Top 5 performance metrics that distinguish the blobs in each GMMs with multiple connections.

	$P0$	$P1$	$P2$
1	server_delay_p75	throughput_mbps	bif_mean
2	server_delay_p50	network_delay_mean	bif_p95
3	throughput_mbps	rtt_mean	bif_p75

Table 5.9: Top 3 performance metrics that distinguish each blob in the GMM when HTTP Keep-Alive is disabled.

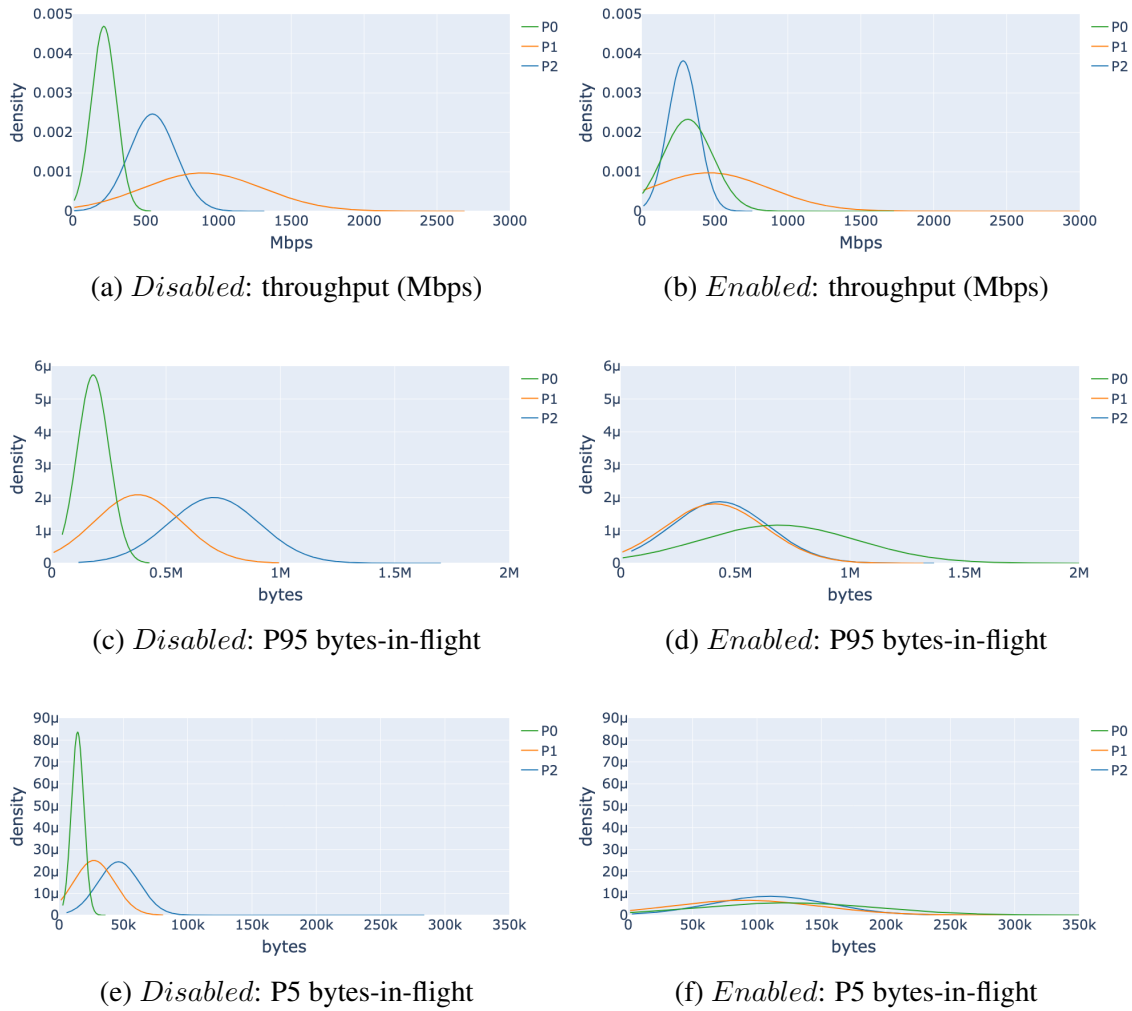


Figure 5.21: Kde plots of each blob in throughput and bytes-in-flight when HTTP Keep-Alive is disabled (*Disabled*: left column) and enabled (*Enabled*: right column).

	avg. no. of RPCs (cross-traffic)	avg. bytes (cross-traffic)	RPC size (byte)	distinguishing info.
P0	41.21	5.266e7	0.54e6	RPC size
P1	30.18	4.028e7	2.08e6	cross-traffic load
P2	41.69	5.512e7	2.41e6	RPC size

Table 5.10: Cross traffic and RPC size information for each blob when HTTP Keep-Alive is disabled with multiple connections.

Before comparing the performance of RPCs with and without HTTP Keep-Alive, we conduct an in-depth analysis of the performance of RPCs in each GMM. The objective of this analysis is to identify the specific performance characteristics and constraints of each blob in both scenarios. When HTTP Keep-Alive is disabled, we use model analysis to determine the most distinguishing performance metrics for each blob. The results of this analysis are summarized in Table 5.9. Additionally, we present kde plots of throughput, and the 5th and 95th percentile of bytes-in-flight in Figure 5.21. The results in Table 5.9 and Figure 5.21 indicate that blob $P0$ has the smallest throughput and bytes-in-flight, blob $P1$ has the highest throughput and the smallest network delay, and blob $P2$ has the highest bytes-in-flight. Furthermore, the GMM analysis in Table 5.10 reveals that the RPCs in blob $P1$ differ significantly in terms of cross-traffic volume, including the number of RPCs and the total volume of bytes transmitted. On the other hand, the RPCs in the blobs $P0$ and $P2$ differ more in their sizes. The low cross-traffic volume experienced by the RPCs in $P1$ may explain their high throughput and small network delay. Despite experiencing similar levels of cross-traffic, the average size of RPCs in $P0$ is almost four times smaller than in $P1$, suggesting that the smaller bytes-in-flight and throughput are due to the overhead of connection establishment and the TCP slow start for each HTTP request/response. In conclusion, when HTTP Keep-Alive is disabled and the network workload is high, the performance of RPCs is mainly influenced by the network, but also depends on the sizes of the RPCs and the presence of cross-traffic. Large RPCs with minimal cross-traffic tend to perform better (blob $P1$), while smaller RPCs are more susceptible to performance issues due to their size and competition for bandwidth with larger RPCs (blob $P0$). This highlights the importance of considering the transfer size when managing network congestion, as smaller transfers may be more vulnerable to performance degradation (56).

Next, we evaluate the network performance of RPCs with HTTP Keep-Alive enabled. Our analysis is summarized in Figure 5.20(b). It shows that network delay is not the only factor affecting the network performance of RPCs when HTTP Keep-Alive is enabled, as the end hosts also contribute to delays in RPCs for blobs $P0$ and $P2$. Table 5.8 indicates that the main differ-

ences in RPC performance among the blobs are RTT and bytes-in-flight. Contrary to when HTTP Keep-Alive is disabled, RPC size is no longer a distinguishing factor among the blobs when it is enabled, as the average size is around 2×10^6 bytes for all three blobs. However, the volume of cross-traffic still plays a crucial role in determining the performance of RPCs, as shown in Table 5.11. RPCs in blob *P1* experience the least cross-traffic, resulting in the smallest network delay and highest throughput. When comparing the cross-traffic information from Table 5.11 with that from Table 5.10, it is clear that cross-traffic increases significantly when HTTP Keep-Alive is enabled.

	avg. no. of RPCs (cross-traffic)	avg. bytes (cross-traffic)	RPC size (byte)
<i>P0</i>	44.76	8.685e7	2.20e6
<i>P1</i>	39.67	7.655e7	1.92e6
<i>P2</i>	44.55	8.557e7	1.92e6

Table 5.11: Average cross-traffic and RPC size of each blob when HTTP Keep-Alive is enabled.

Table 5.12 presents a comparison of the top five performance metrics that distinguish blob *P0* when HTTP Keep-Alive is enabled versus disabled. The table shows that the use of HTTP Keep-Alive results in significant increases in bytes-in-flight and RTT, which are the most significant differences between the two scenarios. These differences are also depicted in Figures 5.21 and 5.20(b). It is worth noting that similar changing trends are observed in blobs *P1* and *P2*, but are not included here for brevity.

	1	2	3	4	5
<i>P0</i>	bif_min	rtt_p5	rtt_p25	rtt_min	bif_p5

Table 5.12: Top 5 distinguishing performance metrics for *P0* with and without HTTP Keep-Alive.

Overall, our test results indicate that increasing the number of connections between each server/client pair from one to five degrades the performance of RPCs when HTTP Keep-Alive is enabled, resulting in increased delay and decreased throughput. This is contrary to the widely held belief that HTTP Keep-Alive improves performance. Further analysis reveals that the volume of cross-traffic increases with HTTP Keep-Alive, as shown in Figure 5.22. The results show

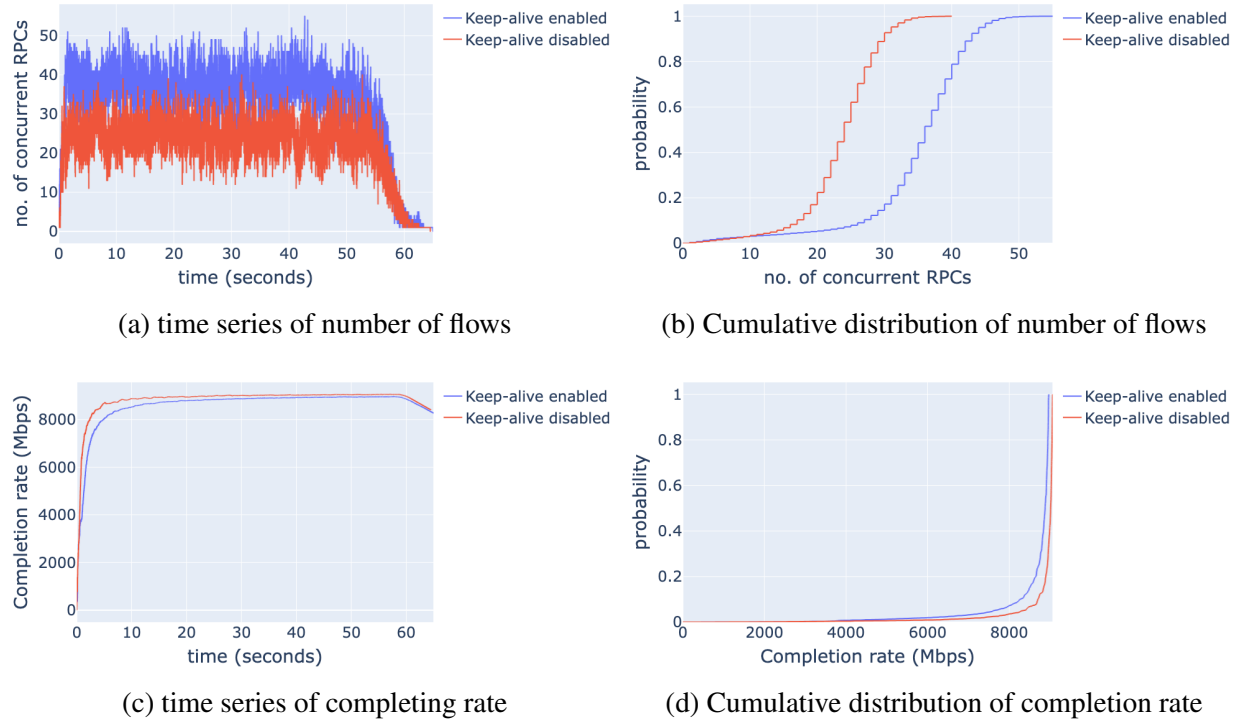
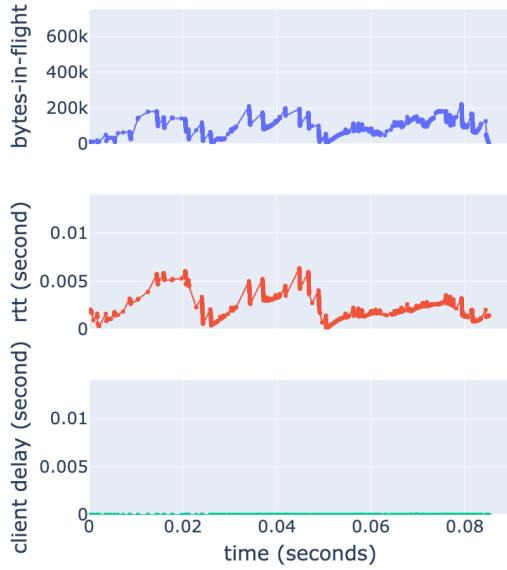


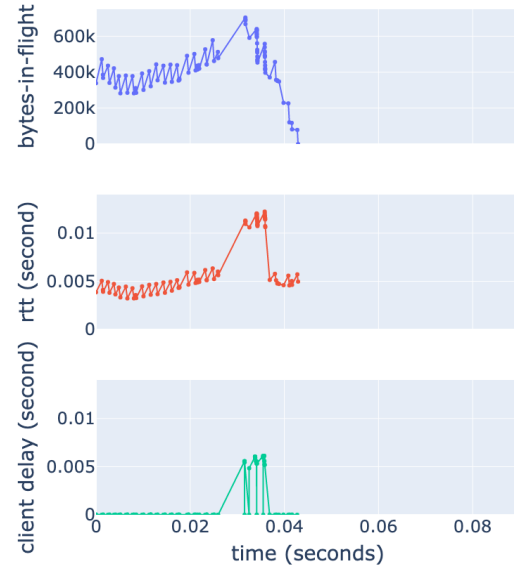
Figure 5.22: Number of concurrent RPCs and completion rate with and without HTTP Keep-Alive.

that when HTTP Keep-Alive is disabled, the median number of RPCs is 25 and the completion rate is around 8.8 Gbps, whereas with HTTP Keep-Alive enabled, the numbers are 35 and 9 Gbps, respectively. Although the total workload is similar, with around 67,000 MB of data transferred in total, the additional overhead of TCP connection establishment and teardown, as well as the slow start stage when HTTP Keep-Alive is disabled, reduces competition in the network. This allows each RPC to acquire a larger share of the available bandwidth resources, resulting in better performance with a higher completion rate, as depicted in Figures 5.22(c) and 5.22(d), and reduced RTT, as evidenced by the values presented in Table 5.6.

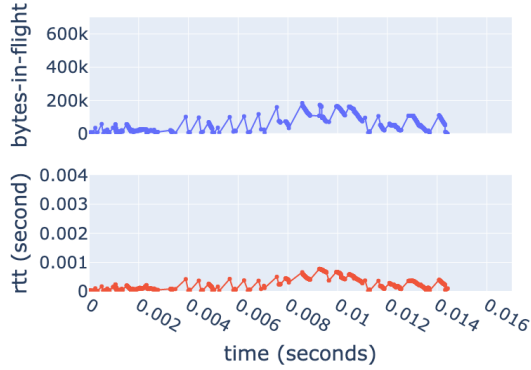
Validating Constraints with Ground Truth Information Finally, using ground-truth information obtained from pcap traces, we present time sequence plots of bytes-in-flight and RTTs for representative RPCs in blobs $P0$ and $P1$ with HTTP Keep-Alive both disabled and enabled. Our GMM analysis shows that, despite increased network and client-side delays, RPCs in $P0$ experience higher throughput and byte-in-flight. Figures 5.23(a) and 5.23(b) illustrate that when



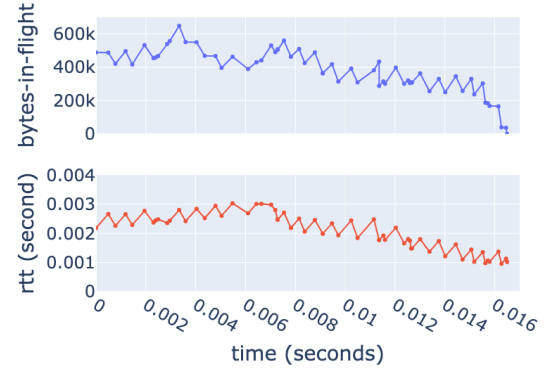
(a) $P0$: without HTTP Keep-Alive



(b) $P0$: with HTTP Keep-Alive



(c) $P1$: without HTTP Keep-Alive



(d) $P1$: with HTTP Keep-Alive

Figure 5.23: Sample time sequence plots of bytes-in-flight and RTTs of example traces in $P0$ and $P2$.

HTTP Keep-Alive is enabled, the example RPC experiences larger bytes-in-flight and RTTs at the beginning of the transfer. Additionally, during the transmission, several bursts in client delay almost double the measured RTT. Despite these increased delays, the transfer duration is reduced by around half, resulting in higher throughput, compared to when HTTP Keep-Alive is disabled.

Our GMM analysis indicates that enabling HTTP Keep-Alive has a negative impact on the performance of RPCs in blob $P1$, as demonstrated by an increase in network delay (Figure 5.20(b)) and a decrease in throughput (Figure 5.21(b)). This conclusion is further supported by the time sequence plots of bytes-in-flight and RTTs for a representative RPC in $P1$, both with and without HTTP Keep-Alive enabled (Figures 5.23(c) and 5.23(d)). The sample RPC shows that, with HTTP Keep-Alive enabled, the transfer begins with high bytes-in-flight and RTTs, which gradually decrease over time. However, this results in a longer transfer duration and lower throughput.

The results of this example show that when competing for bandwidth resources in a congested network, the RPCs in blob $P1$ adopt a more conservative approach, sending less data into the network, while the RPCs in $P0$ become more aggressive, sending more data to increase their share of bandwidth. This balancing of behaviors helps to reduce the disparities in resource allocation, leading to a fairer distribution of resources. However, due to the increased volume of cross-traffic, the overall performance is still degraded compared to when HTTP Keep-Alive is disabled.

5.4.3 Experiment 3: The Impact of Communication Patterns

Lastly, we investigate the impact of different communication patterns on RPC performance when multiple connections exist between each server and client with HTTP Keep-Alive enabled. In contrast to the 1-to-1 communication scenario described in Section 5.4.2.1, where each client communicates with a specific server via multiple TCP connections, we examine a 1-to-5 scenario where each client communicates with five different servers through a single TCP connection. Each connection transferred 1 GB of data through multiple HTTP requests and responses. By utilizing GMM analysis, we observe no notable variations in RPC performance between the two

communication patterns. This finding suggests that, when the network workload is similar, the specific choice of strategy for distributing RPC requests does not significantly influence on the network performance of RPCs in our testbed. For a detailed analysis, please refer to Appendix B.2.

5.5 Use Case 3: Detecting Performance Anomalies

This section presents an example of how GMMs can be used to detect an anomalous server in our testbed. Performance anomalies are common in data centers and can have different causes. However, the proportion of RPCs exhibiting abnormal behavior is usually small compared to normal ones, making it challenging to identify them using time series and distribution plots alone.

The experiment involves five connections between each server and client, with each connection transferring 1 GB of data with HTTP keep-alive enabled. In this case study, the anomaly was not intentionally introduced, but instead was discovered while analyzing the results of a GMM with HTTP keep-alive enabled in an experiment.

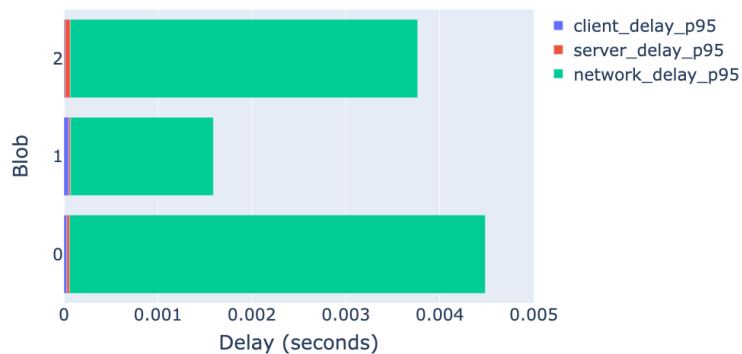


Figure 5.24: Delay breakdown of each GMM blob.

Figure 5.24 illustrates the breakdown of the delay in the network and on the end hosts for each blob in the GMM, with blob 1 experiencing the smallest network delay among the other blobs, resulting in the highest throughput. GMM-based categorical analysis shows that the RPCs in blob 1 are different from the other blobs in terms of machine names. The cosine similarity between the RPCs in blobs 0 and 2 in terms of machine names is as high as 0.99, indicating a

consistent distribution of RPCs among a group of machines. In contrast, the cosine similarity between blob 1 and blobs 0 and 2 is only around 0.29. Further analysis reveals that 48% of the RPCs in blob 1 are from the server s_hp_{104} , while there are almost no RPCs from s_hp_{104} in blobs 0 and 2, as shown in Figure 5.25.

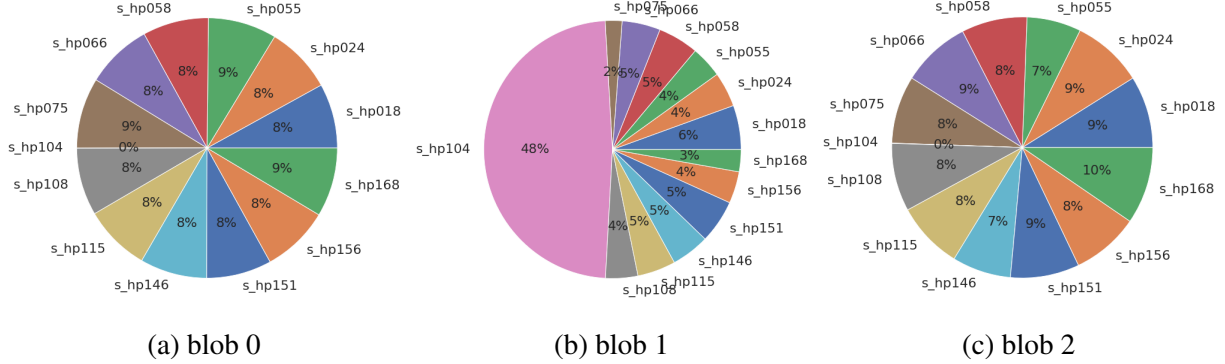
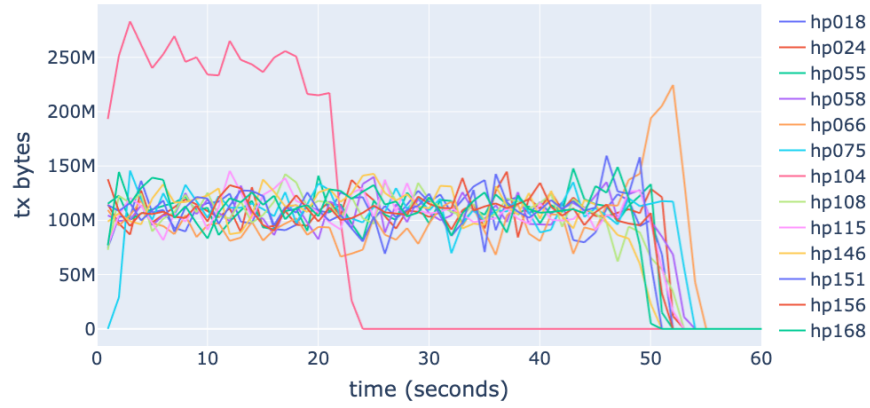


Figure 5.25: The ratio of RPCs from each machine in each blob. "s_" in the machine name indicates servers.

To validate the accuracy of the GMM results, we conduct further analysis on the packet information captured on the end hosts and the overall sending rate of each server. This is done by examining the transmitted packets per second, as collected by the *ifconfig* command, and by analyzing the data from the pcap files. It is important to reiterate that the information collected by *ifconfig* from each end host is not utilized as input for constructing GMMs. Instead, it serves as ground-truth information used solely for validating the GMM results, specifically in cases where there are variations in RPC performance across different end hosts. Figure 5.26(a) displays the time series of transmitted packets (tx_pkts) from each server within an interval of one second. As can be seen in Figure 5.26(a), the server s_hp_{104} transmitted approximately twice as many bytes in the first 25 seconds of the experiment compared to the other servers. This high transmission rate resulted in s_hp_{104} completing the transfer of 5GB of data in half the time, as shown in Figure 5.26(b). These results obtained from *ifconfig* provide evidence that server s_hp_{104} demonstrates "anomalous" behavior characterized by a significantly higher transmission rate. This finding aligns with the conclusions derived from the GMM analysis, which indicates that RPCs from s_hp_{104} exhibit distinct behaviors compared to those from other servers. Unfortunately, the



(a) time series of transmitted bytes within each second from each server from ifconfig

	Mbps	data_bytes	duration
server_node_id			
s_hp018	773.975375	4991103960	51.589279
s_hp024	788.533069	4991759904	50.643506
s_hp055	817.191687	4992838664	48.878017
s_hp058	771.807884	4984095640	51.661516
s_hp066	771.947461	4992085704	51.734979
s_hp075	781.864616	4987575184	51.032622
s_hp104	1580.316686	4989186808	25.256643
s_hp108	798.643867	5000146720	50.086372
s_hp115	774.540575	4994757264	51.589367
s_hp146	756.146197	4991832304	52.813409
s_hp151	751.369288	4996383368	53.197632
s_hp156	788.698594	4984506872	50.559308
s_hp168	799.139469	4995656472	50.010359

(b) sending rate of each server based on pcap information

Figure 5.26: The ground truth statistics based on ifconfig and tcpdump.

underlying cause of this anomalous behavior in $s_{hp_{104}}$ remains unknown, as it did not reoccur in our subsequent experiments.

Without the use of GMMs, it can be challenging to identify an anomalous server in the testbed. This is because the structure of the anomaly is not immediately evident from the kde plot shown in Figure 5.27. Detecting such anomalies is particularly challenging in large-scale production HDCs, where there can be terabytes of data transferred among tens of thousands of machines. Performance anomalies in these environments often occur on a much smaller scale compared to normal behaviors, making it difficult to determine whether RPCs from/to a subset of machines are exhibiting divergent behavior in any performance dimensions.

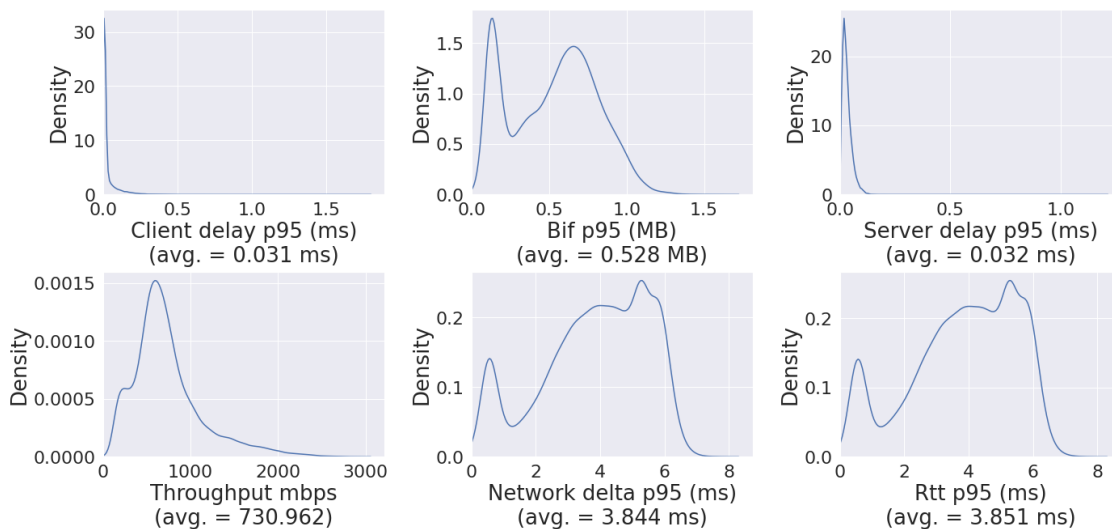


Figure 5.27: KDE plots of delay, rate and volume without GMMs.

5.6 Conclusion

In this chapter, we demonstrate the utilization of GMMs in identifying different network performance experience for RPCs and uncovering their bottleneck constraints in a controlled testbed with traffic patterns that follow the distribution of a representative application in production data centers. The performance metrics for GMM modeling and validation are obtained by extracting statistics from end hosts and switches in the testbed using Simple Network Management Protocol (SNMP), ifconfig, and tcpdump.

In our experiments, we have shown that GMMs are capable of effectively identifying different types of performance experienced by RPCs, despite the disproportional percentages of RPCs experiencing each type of performance and their similar performance in one or more performance dimensions. We validate the effectiveness of GMMs in evaluating the impact of different configuration factors, such as GSO, HTTP keep-alive, and communication patterns, on the network performance of RPCs. The GMM approach provides a more precise “apples-to-apples” comparison when compared to traditional approaches, allowing us to understand the impact of changes on RPCs with different performance. Additionally, we demonstrate the usefulness of GMMs in detecting performance anomalies and pinpointing the most likely causes.

Throughout these experiments, we compared our GMM-based approach with traditional time-series and distribution-based analysis methods. Our GMM-based approach is more effective in uncovering the differences in performance among tens of thousands of RPCs by considering multiple performance metrics simultaneously, such as rate, volume, and latency. The analysis tools introduced in Chapter 3.3 make it easy to interpret the bottleneck constraints affecting RPC performance, enabling us to understand why certain RPCs perform in certain ways. In the following chapters, we will examine real-world case studies conducted in a production HDC.

CHAPTER 6: CASE STUDIES FROM A PRODUCTION HDC

We have implemented the modeling pipeline described in Section 4 in one of the largest production HDCs at Google, and used it for modeling the network performance of real-world applications over the past three years.

In this chapter, we first describe the data instrumentation system in the production HDC for collecting RPC telemetry. We then present several case studies that demonstrate the efficiency of our modeling approach and provide insights into the performance of HDC applications. Each case study involves one or more HDC applications running on different infrastructures or during different time periods. The main objective is to understand how GMM-based analysis reveals the network performance of HDC applications, assesses the impact of infrastructure changes on application network performance, and troubleshoots performance degradation in the production HDC.

6.1 RPC Performance Instrumentation: Fathom

As previously discussed in Section 3.2.2, the RPC library used by applications provides a good vantage point for collecting performance metrics related to local, remote, and network systems. This enables us to analyze and characterize the network performance of applications effectively.

The RPC workflow in the production HDC is similar to that shown in Figures 3.1 and 3.2. For collecting RPC telemetry, we use the existing instrumentation system in the production HDC, Fathom. Fathom passively instruments a sampled RPC and records timestamps when request and response messages traverse each layer in the RPC and network stack – from the shared user RPC library (141) through the kernel TCP/IP and qdisc scheduler to the NIC – in both senders and

receivers to collect performance data for fitting into GMMs. In the next section, we describe the metrics and categorical attributes collected by Fathom.

6.1.1 Performance Metrics Collected

For each instrumented RPC message, Fathom collects the following metrics:

1. Latencies: the timestamps when the request and response messages pass through each layer of the networking stack on the host – from the shared user RPC library, through the kernel TCP/IP and *qdisc* scheduler, and in the network (from when the first byte is sent until the last byte is acknowledged);
2. Rates: sender pacing rate and receiver delivery rate (throughput);
3. Connection state: including the estimated TCP round-trip time (RTT), congestion window size, and number of packets lost during transmission.

All timestamping extensions implemented in Fathom for collecting latencies have been upstreamed to Linux in kernel v3.17 and later (143). As the detailed implementation is beyond the scope of this dissertation, we will not be discussing it here. For collecting rates and connection states, Fathom captures the state of TCP when a timestamp is recorded in kernel. Compared to recording a timestamp when it is received in userspace, this approach has been shown to yield more accurate results by reducing measurement delay, which is particularly important on high-speed networks.

The data collection in the RPC library occurs at the boundary between application-side performance, such as request backlog or processing delays on local/remote end, and network-side performance, such as delivery rate and transfer latency. This enables multiple perspectives on the potential constraints that applications may encounter when executing an RPC transaction. The key metrics captured by Fathom are summarized in Table 6.1, with a detailed description provided. These metrics are then further aggregated (as described in detail in Section 6.1.4) and used as candidates for modeling with GMMs.

Metrics	Description
Rate metrics	
<i>delivery_rate</i>	Throughput of the connection, including <i>delivery_rate_applimited</i> (e.g., when the throughput is limited by application backlog) and <i>delivery_rate_notapplimited</i> .
<i>pacing_rate</i>	Per-connection pacing rate chosen by TCP.
Latency metrics	
<i>pacing_latency</i>	Per-message latency incurred by the pacing layer at the sender.
<i>tx_latency_*</i>	Delta between when the first byte of a message was sent to when we receive the acknowledgment for the last byte. Given the different message sizes, we aggregate the per-message transfer latency in 6 buckets: (0, 1KB], (1KB, 8KB], (8KB, 64KB], (64KB, 256KB], (256KB, 2MB], and (2MB, $+\infty$).
<i>tcp_queueing_latency</i>	Time a message sits on the TCP transmit queue.
<i>rpc_transmit_queueing_latency</i>	Time a message spends in the RPC library buffers after serialization until it is written to the socket.
<i>app_queueing_latency</i>	Time a message spends in the RPC library queues before it is serialized into the binary buffers of the RPC library.
<i>receive_queueing_latency</i>	Delta between when the sender receives the acknowledgement of the message to when the receiver starts processing the RPC.
<i>rate_limit_latency</i>	Per-message latency incurred by the traffic shaping layer at the sender.
Connection state metrics	
<i>min_rtt</i>	Windowed minimum round trip time (RTT) of the connection on a per-acknowledgment basis.
<i>congestion_window</i>	TCP congestion window size.
<i>wire_bytes</i>	Sum of RPC payloads.
<i>message_count</i>	Number of RPC requests/responses.
<i>retx_packets</i>	Number of packets retransmitted.
<i>delivered_packets</i>	Number of packets delivered.
<i>delivered_ce_packets</i>	Number of packets delivered with ECN marks (142).
<i>rwnd_limited_time_ratio</i>	Average ratio that the connection was receive window limited while transmitting an RPC request and response.
<i>sndbuf_limited_time_ratio</i>	Average ratio that the connection was send buffer limited while transmitting an RPC request and response.

Table 6.1: Description of key metrics collected in Fathom.

RPC Metrics Collected by Fathom vs. On Cloudlab (Chapter 5) Despite the common goal of collecting RPC performance metrics, Fathom’s approach in the production HDC and our approach on CloudLab testbed differ notably due to the inherent differences between experimental and production environments. Our approach on CloudLab primarily relies on pcap traces collected from both servers and clients to provide detailed per-packet information, as described in Section 5.2. This enables us to obtain network latency, client latency, and volume metrics related to each packet transmitted in an RPC. In contrast, Fathom only captures a few snapshots of the per-RPC TCP state to obtain rates, network latency, and volume metrics—this is important for managing tracing and storage overhead in production HDCs. Table 6.2 shows when these metrics are collected in Fathom: *rate_limit_latency*, *tcp_queueing_latency*, *pacing_latency*, and *pacing_rate* are logged when the first packet of an RPC is sent, while *delivery_rate*, *min_rtt*, and *congestion_window* are logged when an acknowledgment is received for the last byte sent in an RPC. Given the scale of HDCs, collecting a few snapshots from each of millions of RPCs can still provide informative data for performance monitoring, troubleshooting, and planning, as demonstrated in (39). More fundamentally, as demonstrated in Chapter 5 and later in this chapter, GMMs work well with RPC data collected with both approaches for modeling application network performance.

Snapshot Collection	Metrics
when the first packet is sent	<i>rate_limit_latency</i> <i>tcp_queueing_latency</i> <i>pacing_latency</i> <i>pacing_rate</i>
when the last packet is acknowledged	<i>delivery_rate</i> <i>min_rtt</i> <i>congestion_window</i>

Table 6.2: Fathom metrics and their snapshot collection.

6.1.2 Categorical Attributes Collected

Fathom annotates each instrumented RPC with several categorical attributes related to:

1. Application: including the name of the source and destination user and job. A user is an internal service entity for resource accounting and management, representing a different service. For instance, a user can be a storage service that contains multiple jobs serving other services, such as bandwidth-intensive streaming services or computation-intensive machine learning services¹. In the production HDC, users submit jobs to a cluster manager with customized configurations (107). A job can be configured with different categorical attributes, such as transmission priority and datacenter clusters.
2. Topology: including the location of the source and destination pod, cluster, and metro region. In HDCs, racks are grouped into pods, pods are grouped into clusters, and multiple clusters are interconnected within a metro region. Each pod contains only servers in one cluster, while a cluster may span multiple pods.
3. Transport policies: including transmission priority (QoS class) and congestion control algorithm, which influence transmission.

Table 6.3 shows detailed information about each categorical attribute in Fathom. These categorical attributes are used for analyzing the GMM results, as discussed in Section 4.2.2.

Fathom is a passive monitoring system and does not generate any probing traffic. It covers a significant portion of applications as most applications in HDCs use RPCs. Storing accurate raw distributions for all RPCs would be infeasible due to the enormous amount of data involved. Therefore, Fathom uses sampling and aggregation techniques for maintaining the distribution structure while reducing the processing and data volume burden in the HDC.

6.1.3 RPC Sampling

To efficiently collect measurements with low overhead, Fathom employs a fixed probability random sampling technique. As found in previous research (39), a small sample size, such as one out of thousands of requests, can provide sufficient information for common use cases of tracing

¹In our context, “service” and “user” can be used interchangeably.

Categorical attributes	Description
<i>time</i>	Start timestamp of the aggregation interval.
<i>src_user</i>	Source user name. User is the internal service entity for resource accounting and management. A user can be Google developers and system administrators that run Google's applications and services.
<i>src_job</i>	Source job name. At Google, users submit their work to Borg in the form of jobs (107). A <i>user</i> may contain multiple <i>jobs</i> . Each job consists of one or more tasks that all run the same program.
<i>dst_user</i>	Destination user name.
<i>dst_job</i>	Destination job name.
<i>src_pod</i>	Source pod. A pod is a basic construct of a data-center infrastructure that inter-connects over a thousand hosts to each other and to the data-center network spine.
<i>src_cluster</i>	Source cluster. A cluster may span several pods inside a data-center, but each pod only contains machines for one cluster.
<i>src_metro</i>	Source metro. Multiple data-centers are inter-connected within a metro.
<i>dst_pod</i>	Destination pod.
<i>dst_cluster</i>	Destination cluster.
<i>dst_metro</i>	Destination metro.
<i>QoS_class</i>	Transmission priority of a flow (28). Flows with different <i>QoS</i> are in different queues and served in a weighted round-robin way. The higher the priority, the higher the <i>QoS_class</i> , and the larger the weight.
<i>congestion_control</i>	TCP congestion control algorithm.

Table 6.3: Categorical attributes in Fathom.

data, such as performance evaluation, understanding, and testing. Overall, Fathom monitors billions of TCP connections per second, 24x7, globally across all of Google’s datacenters after sampling.

6.1.4 *t*-Digests: Aggregation that Preserves Distributions

To reduce storage overhead, Fathom aggregates instrumented measurements into distributions based on the categorical attributes listed in Table 6.3 after sampling. Specifically, performance metrics for RPCs with identical attributes are aggregated into one-minute intervals based on the start timestamp of each sampled RPC, which is rounded to the nearest minute. Maintaining accurate distributions, particularly in the tails (i.e., high and low percentiles), is crucial for fitting GMMs with multiple and unknown modes, as well as for diagnosing performance issues that often manifest in the tails of metric distributions (e.g., the 95th percentile of transfer latency). However, storing precise distributions for billions of RPCs per second incurs significant processing and storage overhead due to the massive volume of traffic on HDCs.

Histograms Using histograms is a common approach for reducing storage cost while maintaining distribution information. To construct a histogram from a set of performance measurements, the first step is to divide the range of values into consecutive, non-overlapping bins (or “intervals”) and then count the number of values that fall into each bin.

However, histograms have two weaknesses when used with performance metrics collected in HDCs: (1) Finding an appropriate bin size for each performance metric can be challenging. For instance, rate-related performance metrics can range from $O(100)$ to $O(1e5)$ mbps, while latency-related metrics can vary from $O(1e-3)$ to $O(1)$ seconds. The values may also depend on the application type, underlying infrastructure, and cross-traffic. Using a fixed number of bins may result in the loss of information in some cases. (2) Merging multiple histograms for future analysis can be difficult due to diverse bin sizes.

***t*-Digests** Instead of histograms, Fathom uses *t*-digests (144) for storing metric distributions, as they offer several benefits. First, *t*-digests provide an error bound relative to the quantile, with

higher accuracy at both tails, which is often the most important information for diagnosing performance issues. Second, t -digests have a bounded storage overhead, which is important for handling the large amount of measurements collected in HDCs. Finally, t -digests allow for the aggregation of multiple distributions without loss of accuracy. This enables metrics to be efficiently aggregated at a level of categorical or temporal granularity that is fine enough to capture variations and coarse enough to preserve modal structures, all with a manageable volume of data. Therefore, Fathom aggregates performance measurements listed in Table 6.1 into t -digests at the granularity of a one-minute timescale, based on the the starting time of each RPC and categorical attributes listed in Table 6.3 .

6.1.5 Training Features from Fathom

From these aggregated t -digests of metrics, we describe their distributions using (1) the 1st to 99th percentiles, (2) the differences between two percentiles, such as the difference between the 1st to 99th percentile, and (3) the number of RPC measurements in each t -digest. In total, we extract 313 statistical features, from which we select training features for building GMMs. For confidentiality reasons, all performance metrics are normalized relative to (as a multiple of) the minimum value for each metric in this chapter.

Table 6.4 displays the set of training features that are selected for constructing GMMs, following the approach for feature selection detailed in Section 4.1.1. It is worth noting that metrics related to latency are log-normalized using the equation presented in Eq. 6.1, as indicated by the inclusion of the term “ \log_2 ” in the feature name. For handling zero values and preventing division by zero errors, a constant of $\epsilon = 1e20$ is added to the equation in Eq. 6.1.

$$x' = \log_2(x + \epsilon) \quad (6.1)$$

As outlined in Section 4.1.1, the remaining performance statistics derived from Table 6.1 are not utilized as input for building GMMs. This is because these either (1) do not follow a normal

	Feature metric	Related entity/resource
1	<i>app_queueing_latency_sec_p95_log2</i>	application local CPU/mem transmission policy congestion control algorithm
2	<i>rpc_transmit_queueing_latency_sec_p95_log2</i>	
3	<i>rate_limit_latency_sec_p95_log2</i>	
4	<i>pacing_latency_sec_p95_log2</i>	
5	<i>tx_latency_*_sec_p25_log2</i>	network, application
6	<i>tx_latency_*_sec_p75_log2</i>	
7	<i>tx_latency_*_sec_p95_log2</i>	
8	<i>delivery_rate_mbps_p25</i>	
9	<i>delivery_rate_mbps_p75</i>	
10	<i>delivery_rate_mbps_p95</i>	
11	<i>receive_queueing_latency_sec_p95_log2</i>	remote CPU/mem

Table 6.4: Training features for GMMs. ("tx_latency_*_sec" represents different latency buckets, as described in Table 6.1. In our analysis, we choose 1KB and 8KB.)

distribution, such as the discrete scalar performance metrics including *delivered_packets* and *retx_packets*), or (2) are correlated with already-selected features and therefore do not provide any additional information. For example, the 5th and 50th percentiles of the delivery rate are not selected as they are found to be correlated with the 25th and 75th percentiles, respectively. However, it is important to note that these metrics, particularly these do not follow normal distributions, may prove helpful for analyzing the modeling results in conjunction with the categorical attributes collected in Table 6.3. We will present detailed examples of this in the case studies.

Throughout the remainder of this chapter, we will present several case studies that demonstrate how GMMs can be utilized for achieving the following objectives: (1) Avoid the impact of Simpson’s paradox when interpreting the network performance of applications from large amounts of RPC telemetry data. (2) Gain insights into the performance of a major service. (3) Assess the impact of an infrastructure change on RPC performance. (4) Plan for future infrastructure upgrades. (5) Identify the root cause of performance anomalies.

6.2 Case Study 1: Simpson’s Paradox in RPC Telemetry

In this section, we illustrate several instances of Simpson’s paradox that can occur in performance data, where certain trends may become obscured or even reversed after data aggregation.

Simpson’s Paradox can lead to potentially misleading conclusions when the trend in a set of grouped data differs from the trend in the overall dataset.

Analyzing the hundreds of performance metrics collected from billions of RPCs in HDCs can be challenging due to the diverse nature of these systems. HDCs host numerous applications with diverse service types, communication patterns, traffic volumes, and hardware configurations. For managing the analysis workload, it is necessary to apply certain aggregation schemes. However, determining the “correct” level of aggregation that only combines metrics from the same group to avoid Simpson’s Paradox can be difficult, as it requires a thorough understanding of the hardware configurations and applications involved.

To demonstrate the impact of Simpson’s paradox on RPC performance in HDCs, we examine telemetry data from an application running in a production cluster. The application comprises over 4,000 jobs with RPCs belonging to 4 different QoS classes, distributed across more than 10 pods with different hardware configurations.

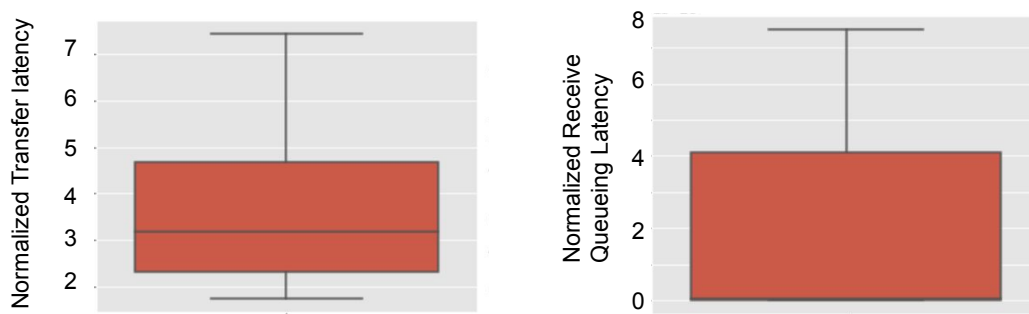


Figure 6.1: Boxplots of the overall transfer latency (left) and receive queueing latency (right) for the application.

Figure 6.1 displays boxplots of the overall transfer latency and receive queueing latency for the application, regardless of job type, traffic class, or RPC size. The normalized values ranging from approximately 2 to 7 and 0 to 8, respectively, between the 5th and 95th percentiles. To further analyze the data, we divide the RPCs into three size-based groups: (1) *small*, with RPCs no larger than 8KB, (2) *medium*, with RPCs between 8 and 256 KB, and (3) *large*, with RPCs larger than 256 KB. However, when these RPCs are aggregated into these three groups as shown

in Figure 6.2, the boxplots for the two metrics vary significantly. The overall median normalized transfer latency is around 3, while it is around 17 for large RPCs and 3 for small ones, a ratio of more than 5 times. For receive queuing latency, the overall value is around 4 in the 75th percentile, while it is around 16×10^9 for small RPCs and 0 for medium and large ones. This indicates that when RPCs of different sizes are aggregated, the performance characteristics specific to each size are obscured, and the overall performance along multiple performance dimensions is dominated by different subgroups. Specifically, the overall transfer latency is mainly influenced by medium and small RPCs, while the receive queuing latency is largely impacted by medium and large RPCs, making it difficult to understand the overall performance and correlate multiple performance dimensions.

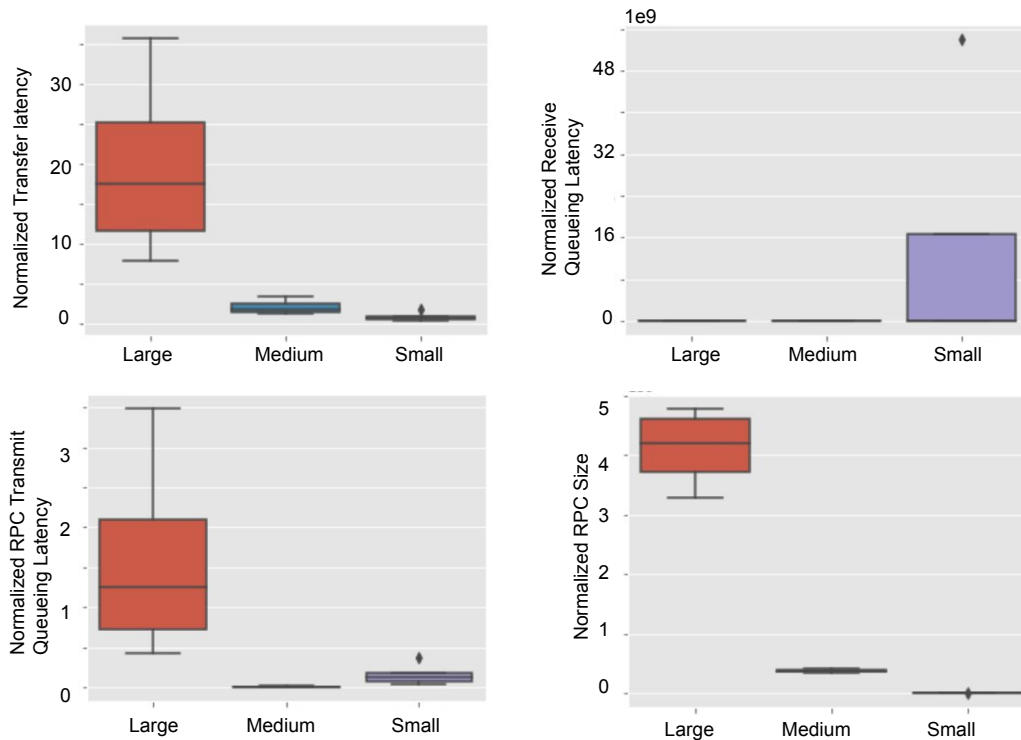


Figure 6.2: Boxplots of transfer latency (top left), receive queuing latency (top right), RPC transmit queuing latency (bottom left), and RPC size (bottom right) for the application after aggregating based on RPC sizes.

In addition to RPC size, there are many other attributes that can impact performance and should be taken into consideration when aggregating RPCs. Figure 6.3 shows boxplots of transfer

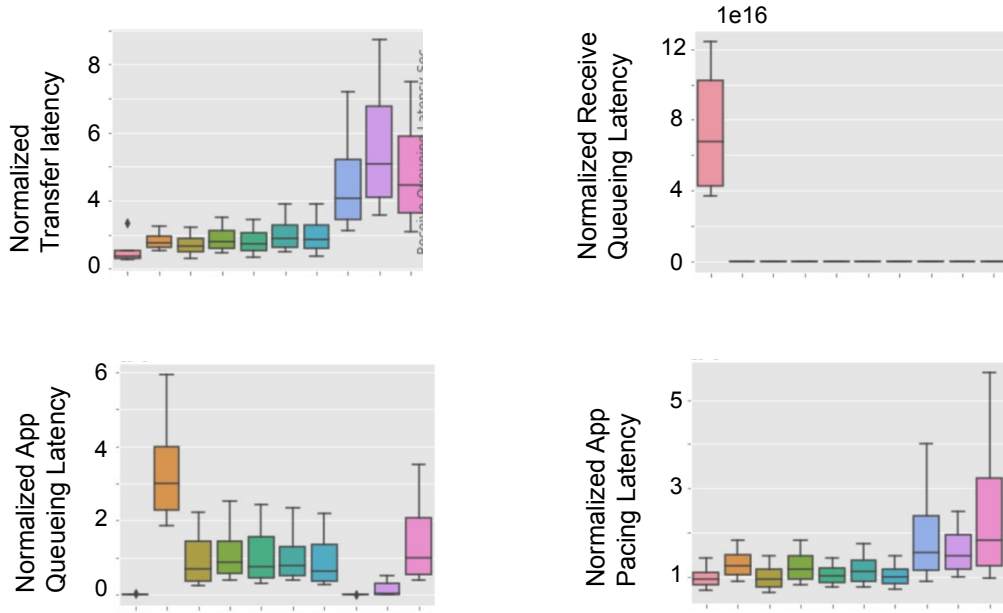


Figure 6.3: Boxplots of transfer latency (top left), receive queueing latency (top right), app queueing latency (bottom left), and pacing latency (bottom right) for 10 different source jobs in this application.

latency, receive queueing latency, application queueing latency, and aggregated pacing latency for different source jobs. As can be seen, latency varies significantly among these jobs. For example, the median receive queueing latency may be as high as $4e16$ for one job, but zero for others, and the app queueing latency may be around 3 for one job but less than 1 for others. Additionally, the performance of these jobs along different dimensions is inconsistent. For instance, the first job has the smallest transfer latency but the largest receive queueing latency, while the second job has the largest app queueing latency. It is crucial to recognize that this information would be hidden if performance metrics were aggregated without considering the types of source jobs associated with each RPC. However, unlike RPC sizes, which can be grouped into discrete buckets, job names are string-based and customized, and cannot be easily discretized. Figure 6.4 shows the transfer and receive queue latency boxplots for 200 jobs from the application, highlighting the variability in performance across different jobs. Given that there are over 4,000 jobs running in the application, it can be challenging to manually analyze each one or to accurately understand how RPCs with different categorical attributes perform and aggregate them before analyzing

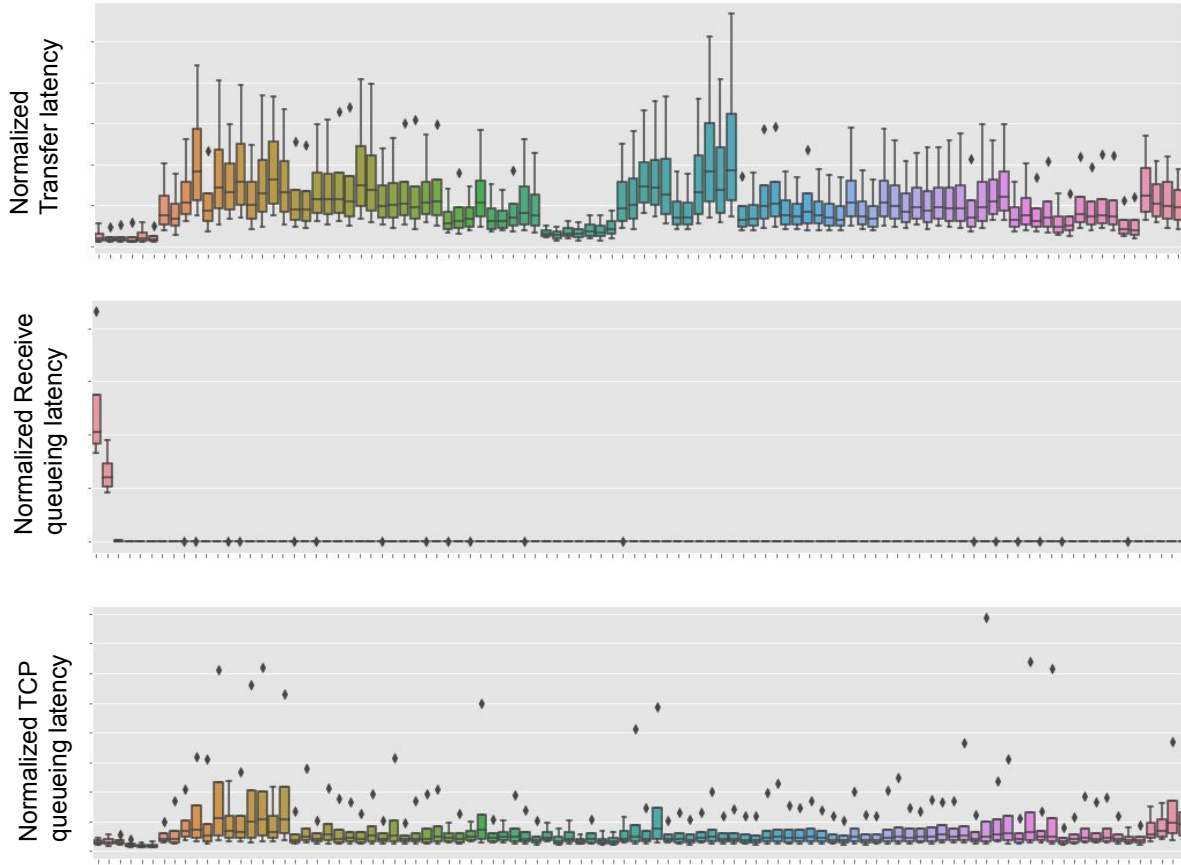


Figure 6.4: Boxplots of transfer latency, receive queueing latency and TCP queueing latency for RPCs from 200 jobs.

their performance. More advanced approaches must be used to automatically identify RPCs that experience similar performance and differentiate those with distinct performance. Below we demonstrate how GMMs can assist us in achieving this goal for this example.

6.2.1 How GMMs Help

Figure 6.2 illustrates that RPC size is a significant factor in determining RPC performance. However, jobs with large RPCs may still exhibit different behaviors on the local/remote host or in the network. In this section, we examine approximately 150,000 *tdigest* aggregated over millions of large RPCs from the production application and demonstrate how GMMs can help group RPCs with similar performance across multiple dimensions and reveal behaviors that were previously obscured. Using the pipeline described in Section 4.2, we group these RPCs into 3

blobs in a GMM: blob 0 comprises approximately 5% of the total RPCs, blob 1 includes almost 35% of the RPCs, and blob 2 contains the remaining 60%.

Figure 6.5 presents transfer latency, RPC queueing latency, and RPC size for each GMM blob. Compared to the overall distribution of large RPCs in Figure 6.2, GMM reveals more distinct behaviors. For example, RPCs in blob 1 exhibit a larger transmit queueing latency compared to others, suggesting a potential local constraint for RPCs in this blob. RPCs in blob 0 have the highest transfer latency and RPC sizes, making them more likely to be network-constrained. By identifying the unique performance of RPCs in each GMM blob, the impact of Simpson’s paradox on the overall distribution becomes clearer. Since 60% of the RPCs experience similar performance (blob 2), the overall distribution shown in Figure 6.2 primarily reflects the experience of this group and ignores the remaining 40%. Therefore, the large transfer latency experienced by RPCs in blob 0, which is about two times larger than that of blob 2, is hidden in the overall distribution. Similarly, the large transmit queueing latency experienced by RPCs in blob 1 is also obscured.

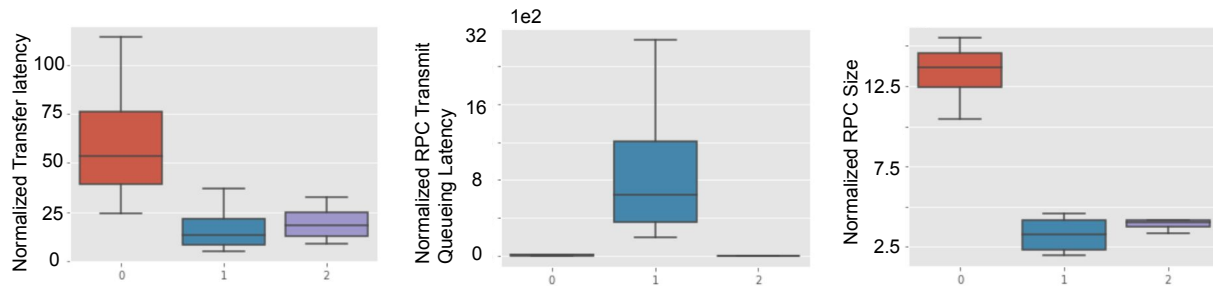


Figure 6.5: Boxplots of transfer latency (left), RPC transmit queueing latency (middle), and RPC size (right) for the application after aggregating based on RPC sizes.

The above examples demonstrate that the amount of information that can be gleaned from performance data in production HDCs is closely tied to how the data is aggregated. An “incorrect” level of aggregation may obscure critical information and result in an inaccurate understanding of RPC performance in HDCs. GMMs can automatically identify distinct RPC performance by simultaneously considering multiple performance metrics, regardless of the traffic volume of RPCs with each performance, and minimize the impacts of Simpson’s paradox. As a result, dif-

ferent optimization techniques can be applied to improve RPCs with different performance and constraints.

6.3 Case Study 2: Evaluating the Performance of a Major Service

In this case study, we evaluate the performance of a storage infrastructure service, known as *StoreService*, in the production HDC network. *StoreService* operates in many Google’s datacenters globally and is widely used by many applications to store immutable data. Specifically, we examined one week’s worth of Fathom measurements for *StoreService* running in one datacenter in mid-2019. During this period, the service accounted for nearly 20% of the entire traffic inside the datacenter. *StoreService* has 3 main jobs that communicates with 17 other destination users (clients or backend servers), and uses multiple QoS classes. In total, we sampled several millions of RPCs, aggregated into 45,945 one-minute measurements with different categorical attributes,². A GMM was then built using this data.

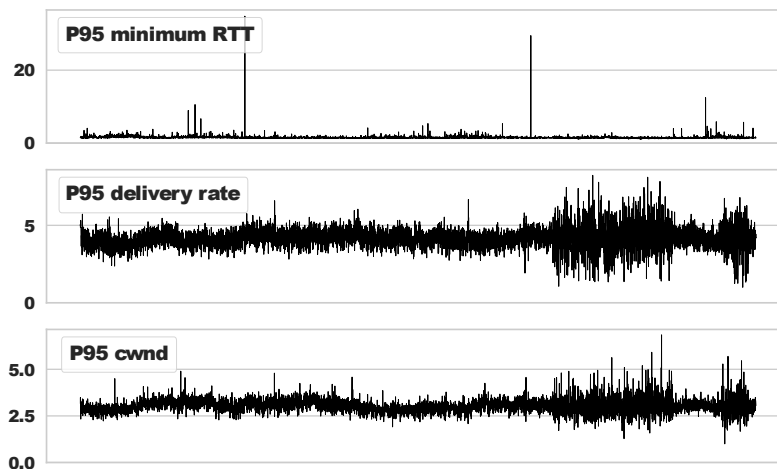


Figure 6.6: *StoreService*: Time-series of normalized 95th percentiles of minimum RTT, cwnd, and delivery rate

Figure 6.6 shows the time series plot of the normalized 95th percentiles of the minimum RTT, congestion window size (cwnd), and delivery rate for the week-long RPC measurements from

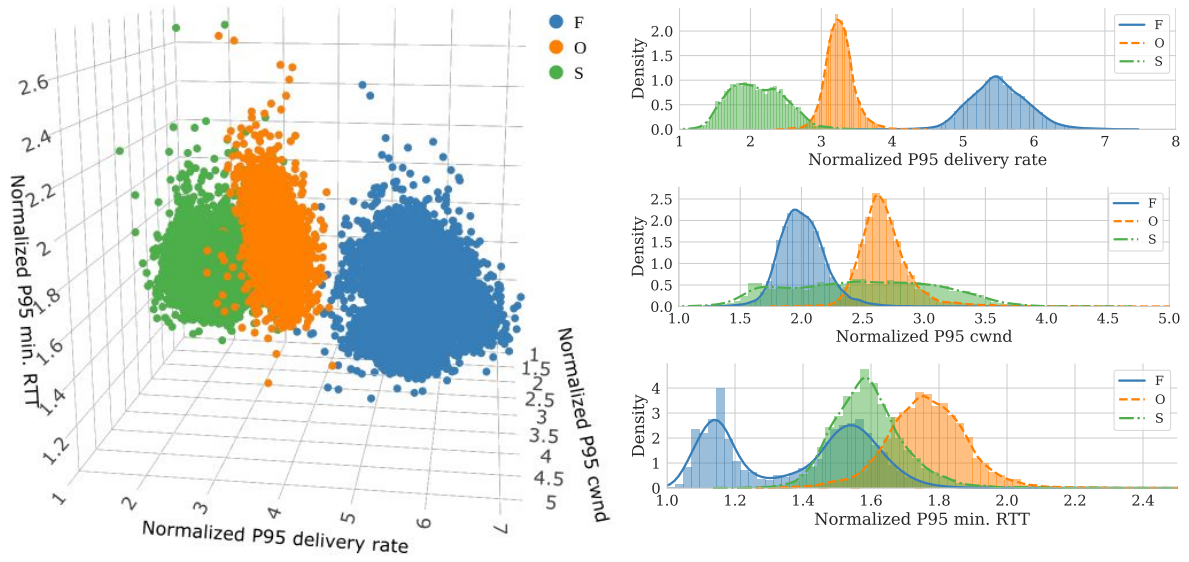
²Note that not every combination of categorical attributes has sampled RPCs in each minute because (1) applications may not run 24/7 in HDCs, and (2) sampling in Fathom.

StoreService. As can be seen, all three metrics exhibit large variations and noticeable fluctuations. For example, the 95th percentile of the delivery rate typically ranges from 3 to 5 times, with occasional fluctuations reaching around 8 times. The data volume is large, and manual analysis to interpret the data would be prohibitively time-consuming.

6.3.1 GMM Results

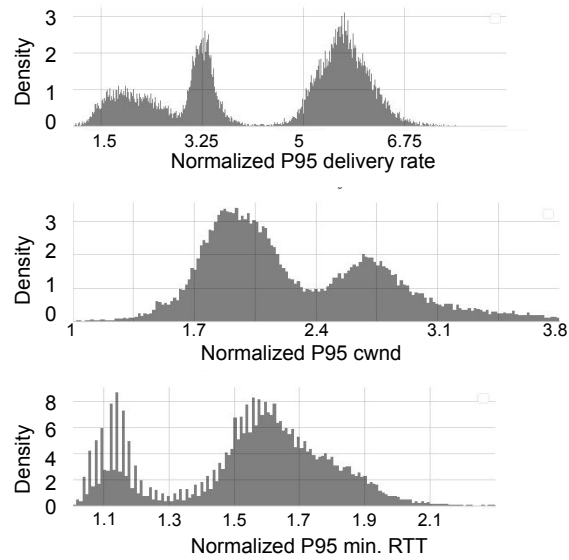
A GMM was trained on the 45,945 measurements and resulted in three blobs, denoted as \mathcal{O} , \mathcal{F} , and \mathcal{S} , each of which is a multivariate Gaussian distribution. Figure 6.7 includes a 3D scatter plot (6.7(a)) and density plots (6.7(b)) of three specific normalized metrics: the 95h percentile of the delivery rate, the 95h percentile of cwnd, and the 95h percentile of the minimum RTT. We observe that:

- The axes in Figure 6.7(a) represent different types of constraints (rate, volume, and latency) that applications can experience in the network. The measurements within each blob are closely coupled across multiple performance metrics. For example, the RPCs in the blob \mathcal{F} experience the highest 95th percentile of the delivery rate and the smallest 95th percentile of cwnd. This coupling is not obvious when looking at individual metrics without blob breakdown, as shown in Figure 6.7(c). To understand how RPCs perform across multiple dimensions, it is necessary to dissect the data accordingly or use the bursting and linking techniques described in Section 5.3.1.
- The three blobs occupy different regions in the 3D scatter plot, reflecting how their performance is impacted differently by the underlying constraints (discussed further later).
- Within each blob, several different metrics are well modeled by Gaussian distributions, as further supported by Q-Q plots in Figure 6.8. The distribution of the 95th percentile of the minimum RTT of the RPCs in blob \mathcal{F} is bimodal as shown in Figure 6.7(b). Occasionally, new modes may emerge at higher RTT percentiles, but unless there is support for them in other dimensions, the model will create a broad, flat, Gaussian distribution that covers the



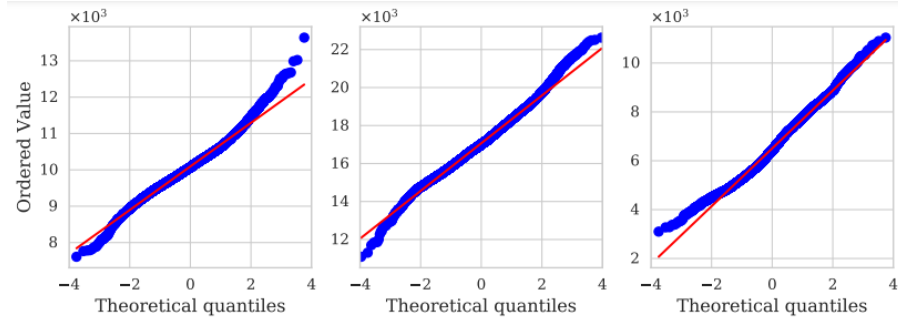
(a) 3D scatter plot

(b) Density plot of each GMM blob.

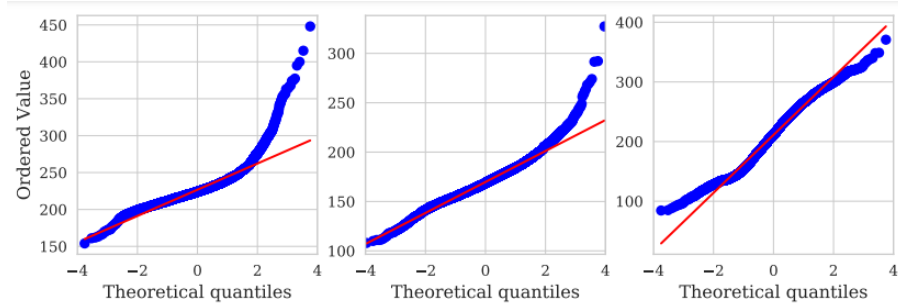


(c) Density plot without blob breakdown.

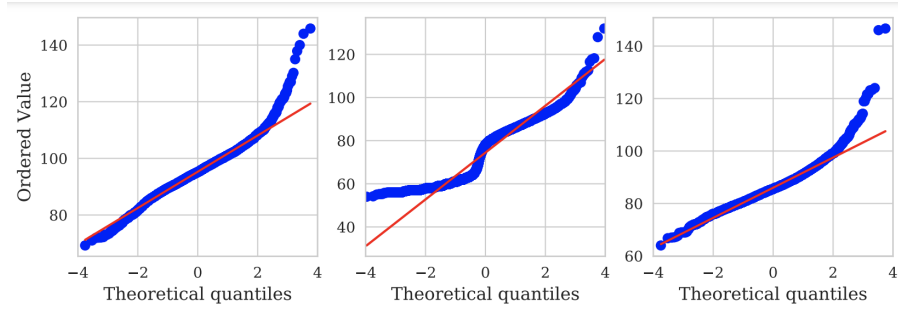
Figure 6.7: GMM Modeling: 95th percentiles of delivery rate, congestion window size, and minimum RTT.



(a) 95th percentile of delivery rate



(b) 95th percentile of cwnd



(c) 95th percentile of minimum RTT

Figure 6.8: Q-Q plot of measurements in each blob (from left to right: O , F and S).

data. In this case, the bimodal distribution is desired because splitting out a mode that only exists in one or two dimensions can create a mess in all the other dimensions. For example, the distribution of the 95th percentile of the delivery rate and cwnd of the RPCs in blob \mathcal{F} is unimodal (Figure 6.7(b)). Therefore, it is preferable for the model to maintain a bimodal distribution in the 95th percentile of the minimum RTT and to further examine them at a finer level in Section 6.4.

The results also suggest that the distribution of the delivery rate fits better to a Gaussian distribution in both tails compared to cwnd, which is a discrete parameter, and the minimum RTT, which is more susceptible to noise from queuing delays.

This case study demonstrates the use of GMM for analyzing a substantial amount of performance data from a production HDC application. The data consists of diverse jobs, QoS classes, and destination users. By distilling the data into a small number of Gaussian distributions, each representing a unique performance behavior, GMM significantly reduces the dimensionality of data. This reduction enables the analysis of tens of thousands of performance metrics and millions of RPCs using only a few operating points. The GMM analysis identifies three distinct performance behaviors in the network performance of the *StoreService*, each with different rates and latencies. These results provide deeper insights and demonstrate the usefulness of GMM in interpreting performance data in HDCs.

6.3.1.1 GMM Distinguishes Performance Constraints

Figure 6.7(a) demonstrates that the delivery rate for \mathcal{F} is higher than other blobs. However, it is not feasible to visually inspect and compare hundreds of metrics in this way. Therefore, we proceed to quantify the differences in the distributions of the 313 performance metrics captured in our measurements for the three blobs.

To quantify the differences between the distributions of a given metric for two blobs, we use the Wasserstein distance as described in Chapter 4. Specifically, we compute the 2-Wasserstein distance between each pair of blobs based on the mean and covariance matrix constructed by

Blob Pairs	$(\mathcal{O}, \mathcal{F})$	$(\mathcal{O}, \mathcal{S})$	$(\mathcal{F}, \mathcal{S})$
2-Wasserstein distance	21.17	22.62	19.58
Metric 1	<i>delivery_rate</i>	<i>pacing_rate</i>	<i>delivery_rate</i>
Metric 2	<i>cwnd</i>	<i>delivery_rate</i>	<i>pacing_rate</i>
Metric 3	<i>pacing_rate</i>	<i>tx_latency</i>	<i>tx_latency</i>

Table 6.5: The most distinguishing metrics between blob pairs.

GMM for each blob. This distance indicates the overall difference in RPC performance between the two blobs. Next, we calculate the 1-Wasserstein distance for each performance metric between GMM blobs. However, to avoid listing the distance for each metric between any two blobs, which would result in 48,672 distances ($312 \times 312 / 2$) between two GMM blobs, we summarize the average distance based on the base metric they are derived from (such as delivery rate, RTT, cwnd, and queuing latency). For example, to calculate the average distance of the base metric delivery rate, we average the distances between all statistics (as described in Section 6.1.4) derived from the delivery rate. Table 6.5 presents the three most distinguishing base metrics with the largest average 1-Wasserstein distance for each pair of blobs, and Figure 6.9 shows the boxplots for these metrics. Note that we do not include the boxplot of the pacing rate, as the trend is consistent with the delivery rate. Our findings are as follows.

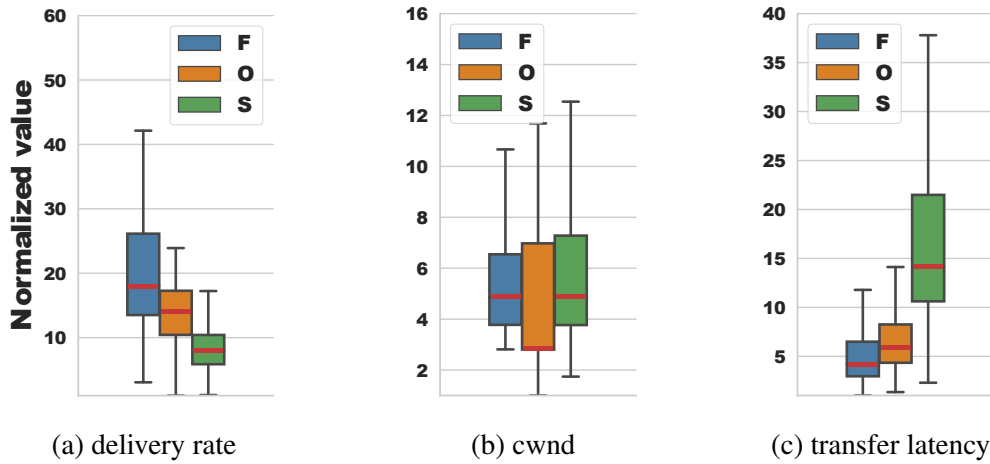


Figure 6.9: Boxplots for the three most distinguishing metrics.

- \mathcal{F} is least constrained by the network. Table 6.5 shows that the most distinguishing metric for \mathcal{F} against the other blobs is the delivery rate. Figure 6.9 supports this finding by

demonstrating that all percentiles of the delivery rate observed in \mathcal{F} are significantly higher than those for the other two blobs. Specifically, the median delivery rate in \mathcal{F} is around 2.2 times and 1.3 times larger than in \mathcal{S} and \mathcal{O} , respectively.

Furthermore, Figures 6.9 and 6.7(b) indicate that the transfer latency and the 95th percentile of minimum RTT observed in \mathcal{F} are significantly lower than in the other two blobs. This suggests that the RPCs in \mathcal{F} experience low queueing delays and achieve a high delivery rate, indicating that they are the least constrained by network bandwidth compared to the other two blobs.

- \mathcal{S} is most constrained by the network. The delivery rate and transfer latency are the two most distinguishing metrics that distinguish \mathcal{S} from the other blobs. The delivery rate for RPCs in \mathcal{S} is much lower than in \mathcal{O} and \mathcal{F} , while the transfer latency is significantly higher.

Our analysis shows that comparing the most distinguishing performance metrics across different GMM blobs helps us gain insights into the RPC experience and constraints in HDCs.

6.3.1.2 Categorical Attributes Influence the Performance Behavior of a GMM Blob

While the previous section provided insights into the performance of RPCs in each blob, these are not enough for developers to determine how to improve their performance. To provide more actionable results, we need to understand what a blob is composed of by characterizing it with attributes that are meaningful to developers. To achieve this, we study the categorical composition of each blob based on the nine different categories captured in Fathom, as previously discussed in Chapter 4.2 and Table 6.3.

	<i>QoS class</i>	<i>Job</i>	<i>Dst. user</i>
\mathcal{O}	0	0.002	0
\mathcal{F}	0.001	0.366	0.680
\mathcal{S}	0	0.186	0.004

<i>Blob pair</i>	<i>QoS class</i>	<i>Job</i>	<i>Dst. user</i>
$\mathcal{O} \ \mathcal{F}$	0	0.991	0
$\mathcal{O} \ \mathcal{S}$	0	0.001	0
$\mathcal{F} \ \mathcal{S}$	1	0.007	0.831

a Entropy

b Cosine Similarity (*cs*)

Table 6.6: Categorical analysis of blobs.

We first identify the most distinguishing categorical attributes for each pair of blobs for the *StoreService* application through the GMM analysis described in Section 6.6. These attributes include the QoS class, job, and destination user as shown in Table 6.6. A destination user may be a client user accessing the storage service or another service that relies on *StoreService*, such as image and monitoring services, with different QoS classes. Our findings can be summarized as follows:

- Table 6.6a shows that RPCs in each blob primarily belongs to a single QoS class ($entropy(QoS\ class) \approx 0$). Additionally, the QoS class of RPCs in \mathcal{F} and \mathcal{S} is the same ($cs(QoS\ class, \mathcal{F}, \mathcal{S}) = 1$), but differs from the QoS class of RPCs in \mathcal{O} . After checking the QoS values for the majority of RPCs in the respective blobs, we find that the QoS class of \mathcal{O} has the lowest priority for scheduling bandwidth in network switches.
- Table 6.6b shows that RPCs in \mathcal{F} and \mathcal{S} communicate with almost the same set of destination users ($cs(dst.\ user, \mathcal{F}, \mathcal{S}) \approx 0.8$) and use the same QoS class, but belong to different sets of jobs ($cs(job, \mathcal{F}, \mathcal{S}) \approx 0$). As previously seen in Figure 6.9, RPCs in \mathcal{S} experience a much lower delivery rate and higher transfer latency than in \mathcal{F} . Upon further analysis of the data for *StoreService*, we find that the average RPC payload size (*wire_bytes* in Table 6.1) for jobs in \mathcal{F} is approximately two times higher than in \mathcal{S} . This suggests that when RPCs in \mathcal{F} and \mathcal{S} are competing for access to network bandwidth in the same priority queue, the large volume of RPCs generated by \mathcal{F} can cause congestion with RPCs from \mathcal{S} .
- Figure 6.9 also shows that \mathcal{S} has a much lower delivery rate and a much higher transfer latency compared to \mathcal{O} , which is surprising given that RPCs in \mathcal{S} belong to a higher QoS class than in \mathcal{O} . Upon further examination of the data, we find that the average RPC payload size in \mathcal{O} is much larger (~ 27 times) than in \mathcal{S} . Therefore, even though \mathcal{O} has a lower QoS class, it achieves a higher delivery rate than \mathcal{S} . This is expected because a connection can achieve higher throughput if it has a larger volume of data to send when the sending rate is controlled by TCP.

The categorical composition analysis provides actionable insights for developers. For example, we can see that the delivery rate of RPCs in \mathcal{S} is slower than \mathcal{F} even on a similar network path, which may be due to the smaller traffic volume and RPC size of \mathcal{S} . Since these blobs use the same QoS class but belong to different jobs, one potential improvement would be to lower the QoS class of \mathcal{F} to improve the performance of \mathcal{S} . We can also see that \mathcal{O} has a large RPC size but the lowest QoS, which may suggest that the low congestion window size (cwnd) is due to congestion within the service itself, rather than the other two blobs. Another possibility is that the congestion control algorithm is overly conservative when interpreting congestion signals, resulting in an unnecessary low cwnd.

This second case study illustrates the effectiveness of using GMMs to analyze large amounts of data on RPC performance. The GMMs can condense the measurements from hundreds of thousands of diverse RPCs into a small number of blobs, each representing a distinct pattern of network behavior and following a multivariate Gaussian distribution. Importantly, the RPC measurements within each blob have distinct physical meanings, including the rate at which RPCs are transmitted (delivery rate), the time required for each packet to be acknowledged (RTT), and the volume of traffic in flight (congestion window). By analyzing the categorical composition of each blob, actionable insights can be gained to improve the performance of RPCs in each blob.

6.4 Case Study 3: Understanding Impact of Infrastructure Change: Congestion Control

This case study employs our modeling pipeline for investigating the impact of an engineering change made to the HDC on the performance of *StoreService*. Specifically, in mid-2019, a new congestion control algorithm (*newCC*) was introduced to replace the old algorithm (*oldCC*). One of the key features of *newCC* is its conservative approach for adjusting the congestion window, with the aim of reducing packet losses and queuing latencies. Using our pipeline, our objective is to understand the effects of this change on the network performance of *StoreService*.

6.4.1 Background: TCP Congestion Control

The Transmission Control Protocol (TCP) is a transport protocol used for exchanging data and messages between devices and application programs over an internetwork. It provides several services, including reliable data delivery, flow control, and congestion control – we focus on the latter here. Network congestion can occur during transmission if the sender generates too many packets for the network to handle, resulting in degraded performance, such as excessive packet delay, loss, and retransmission. TCP uses a congestion control algorithm to regulate the flow of data and prevent or alleviate congestion. It relies on a congestion window (*cwnd*) and a congestion policy to determine the number of bytes that senders can transmit at any given time. Over the past few decades, several congestion control algorithms for HDCs have evolved for handling the increasing traffic volume and diverse communication patterns.

6.4.2 The Picture Without GMM Analysis

StoreService is a major service in the production HDC that is utilized by many other services. The performance of these services can differ due to their unique traffic patterns, volumes, and QoS priorities. Our experience has shown that the performance of even the same application can vary significantly depending on the time of day or week (e.g., *nighttime* vs. *daytime*, or *weekend* vs. *weekday*). The impact of changing the congestion control algorithm on such diverse traffic may itself be diverse. Therefore, aggregating *all* RPCs during analysis may obscure important trends within certain sub-populations, or even produce misleading trends due to Simpson’s paradox (37). For instance, the gray line in Figure 6.10 shows the normalized median delivery rate of aggregated RPCs before and after deploying *newCC*. The graph reveals several spikes with high fluctuations when considering the aggregated data.

To avoid drawing incorrect or incomplete conclusions, it is necessary to identify the appropriate way to segment the data by grouping RPCs that exhibit similar performance and separating RPCs for which performance is affected differently. Finding an ideal segmentation requires in-depth domain knowledge and significant hypothesis testing when performed manually, which

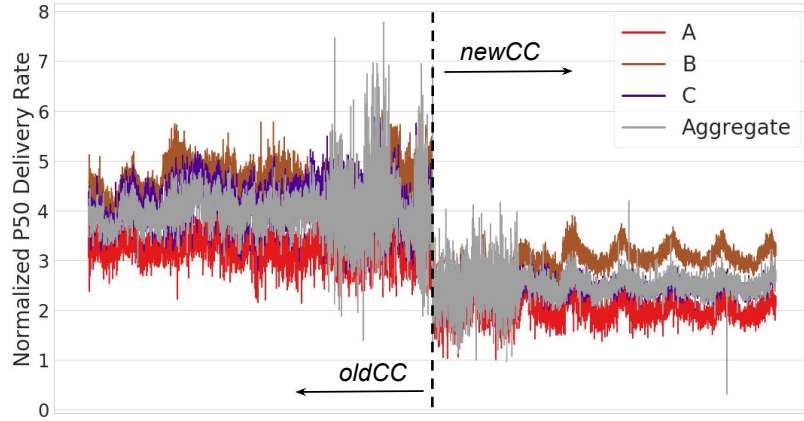


Figure 6.10: Time series plots of the median delivery rate one week before and after deploying *newCC*.

can be time-consuming due to the vast amount of traffic and diverse applications supported by *StoreService*. As demonstrated below, our GMM-based analysis approach addresses this by aggregating based on a joint consideration of multiple metrics and the manifestation of performance constraints.

6.4.3 How Does GMM Help Understand Performance *Before* the Infrastructure Change?

We analyze week-long *Fathom* measurements from a major job of interest (*J*) from *StoreService* running in one datacenter. The data was collected for six weeks, with three weeks before and three weeks after replacing *oldCC* with *newCC*. Each week produced more than 47 million sampled RPCs, which were aggregated into more than 30,000 1-minute *t*-digests.

First, we analyze the *before* data using our modeling pipeline. The pipeline produced three GMM blobs from each week-long data, each with unique performance characterized by three key performance metrics: *cwnd*, *delivery rate*, and *minimum RTT*. Figures 6.11(a)-6.11(c) show the 3D scatter plots of the medians of these metrics for each blob. Our analysis yields several interesting findings:

1. Each of the three blobs is characterized by a unique performance signature. Blob A, for example, is the most bandwidth-constrained, as indicated by its small delivery rate and large minimum RTT. Blob B has the highest delivery rate and may have been constrained

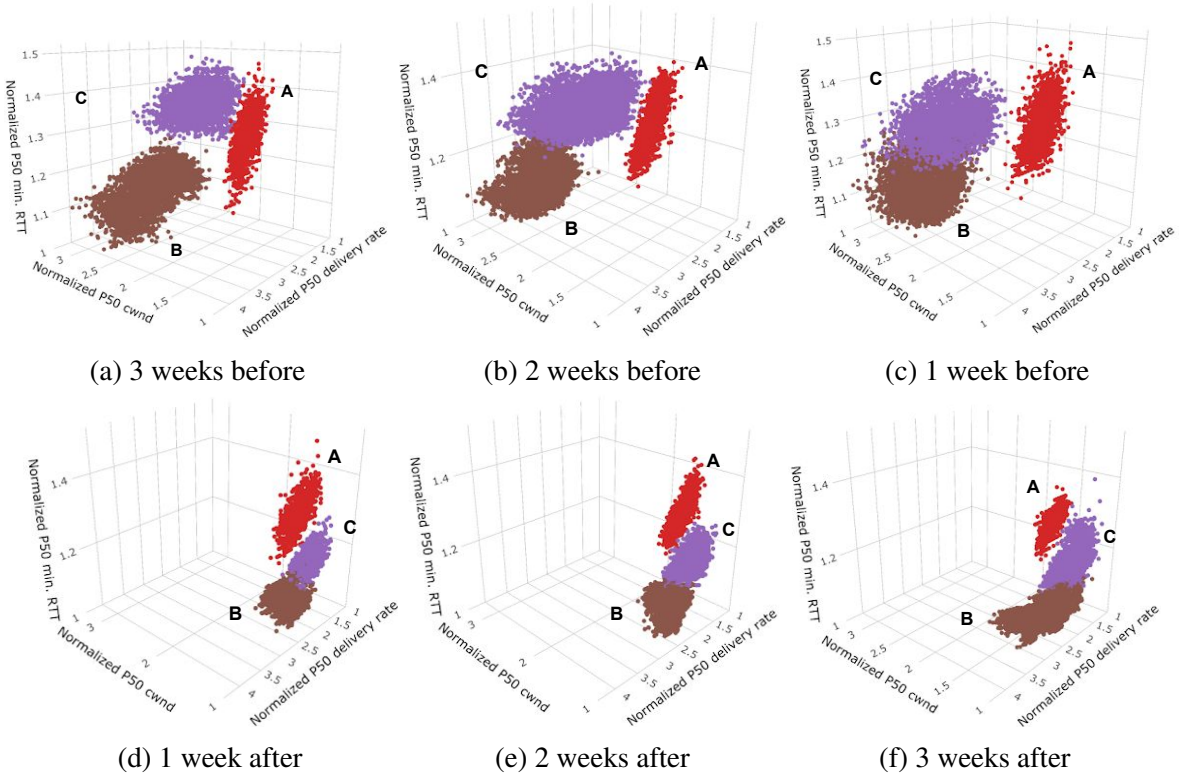


Figure 6.11: 3D scatter plots of RPCs from job J in *StoreService* in the three weeks before and after the deployment of *newCC*.

by either network bandwidth or the application’s data generation behavior (the change in behavior of blob B *after* deployment of *newCC* helped us determine which).

2. The underlying structures of the data are fairly consistent and stable over time. Specifically, the model we used to process the data for each week consistently identified three blobs in a multidimensional space, with relatively stable locations for each blob. We also use categorical analysis to confirm that the categorical composition of each individual blob is consistently stable across the three weeks.
3. The categorical compositions of the blobs differ significantly from each other. For instance, blob A has lower QoS priorities throughout the HDC, which may explain its lower delivery rate despite also having a small minimum RTT (like blob C).

These findings collectively suggest that *oldCC* is likely to favor RPCs with higher QoS priorities (blobs B and C), leading to these RPCs occupying a larger share of bandwidth resources

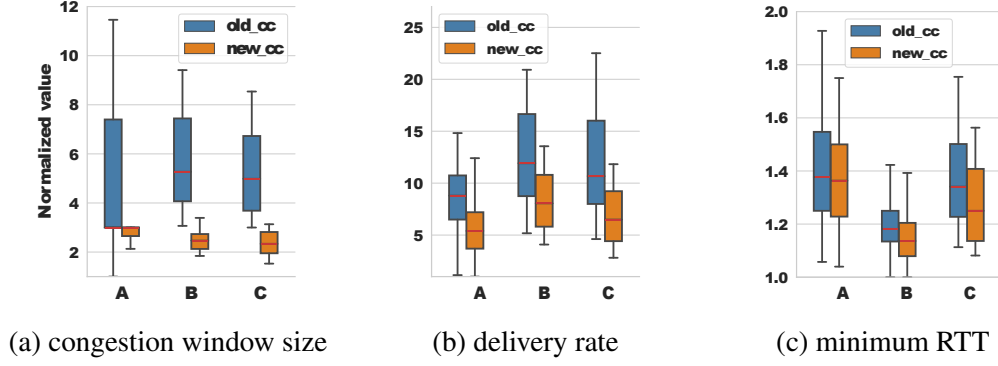


Figure 6.12: Boxplots for the most distinguishing metrics.

compared to RPCs from lower-priority QoS classes. Figure 6.10 depicts the time series of the normalized median delivery rate of RPCs in each blob before and after the deployment of *newCC*. Through GMM analysis, we can see that the performance of RPCs in blob A is finally distinguishable. Additionally, we find that the fluctuations in the performance of RPCs in each blob, as measured by different metrics, are smaller than the aggregate trend suggests. This is because the aggregate trend includes RPCs with different performance characteristics.

Our results demonstrate that the network performance of RPCs in each blob, despite being diverse, adaptive, and complex, exhibits stable structures over periods of several days/weeks. GMM analysis consistently identifies and separates these structures without requiring prior knowledge, significantly accelerating and aiding the process of succinctly interpreting performance.

6.4.4 How Does *newCC* Impact Networking Constraints Experienced by *J* in *StoreService*?

	<i>newCC</i> : A	<i>newCC</i> : B	<i>newCC</i> : C
<i>oldCC</i> : A	7.74	23.13	15.03
<i>oldCC</i> : B	26.11	2.78	6.94
<i>oldCC</i> : C	18.81	5.76	4.02

Table 6.7: The 2-Wasserstein distance between blobs

Next, we apply GMM to model RPCs during three weeks after the deployment of *newCC*. Figures 6.11(d)-6.11(f) show the 3D scatter plots for the GMM blobs in the same dimension

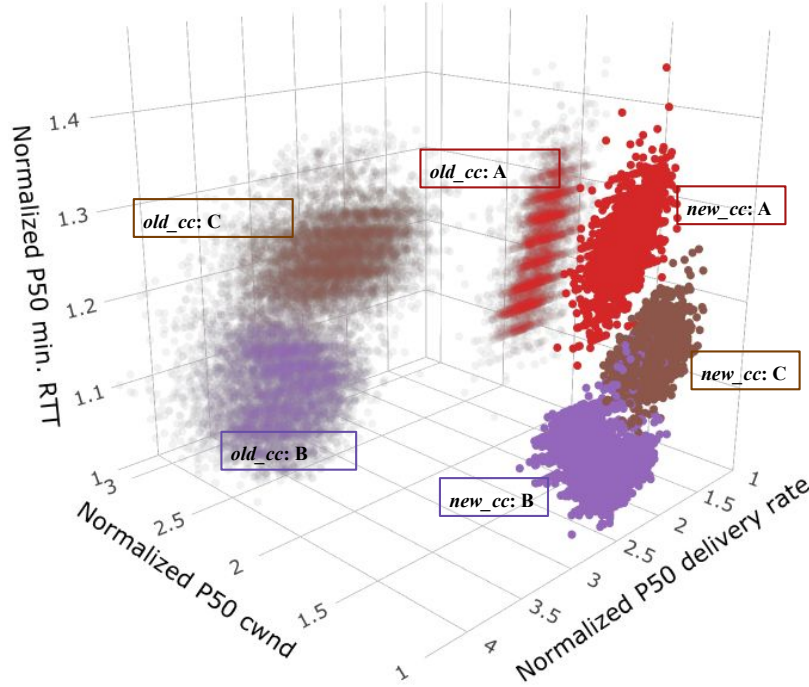


Figure 6.13: *StoreService*: 3D scatter plot—before and after

space as Figures 6.11(a)-6.11(c). The three blobs maintain relatively stable locations in the 3D space over the three weeks. However, when comparing the modeling results from one week before (6.11(c)) and one week after (6.11(d)) the deployment, we see that the blob locations change substantially. For better visualization, Figure 6.13 shows the 3D scatter plots before and after the congestion control change side by side. We can observe that the GMMs reflect the underlying engineering change in the form of noticeable shifts in the otherwise stable blobs. In the following analysis, we will examine the details of this impact.

Matching Blobs To understand the impact of *newCC* on the network performance of RPCs in each blob, it is important to identify the corresponding blobs in the before- and after- GMMs (*e.g.*, Figure 6.11(c) *vs.* 6.11(d)) for an “apples-to-apples” comparison. As described in Chapter 4, we identify corresponding blobs by matching those with the most similar categorical composition and the smallest 2-Wasserstein distances. Table 6.7 lists the 2-Wasserstein distance between GMM blobs before and after the change: corresponding blobs are matched based on the small-

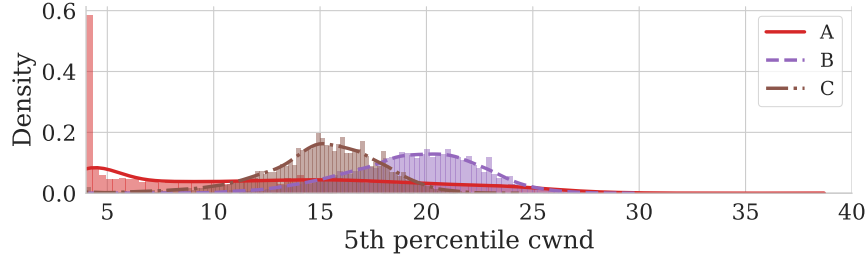


Figure 6.14: Distribution of 5th percentile of cwnd with *new_cc*.

est distance and plotted using the same color in Figure 6.11. We also used cosine similarity to validate the matching process and the results are consistent with Table 6.7.

Ranking the Most-Impacted Metrics Based on the Wasserstein distance, the most distinguishing metrics after deploying *newCC* continue to be *cwnd*, *delivery rate*, and *minimum RTT* (Figure 6.12). Our findings show that: (1) *Cwnd* is significantly lower in all blobs with *newCC*, which aligns with its intended design of being more conservative in adjusting cwnd to reduce packet losses and latency. (2) The decrease of *cwnd* with *newCC* leads to a tighter limit on the delivery rate, and more importantly, reduces the disparity in the delivery rate across blobs. This protects the performance of RPCs from low-QoS classes and suggests that *newCC* has a large impact on RPCs from higher QoS classes by reducing their delivery rate more significantly to achieve a more fair share of bandwidth resources across different QoS classes. Given that *newCC* is designed to be more conservative than *oldCC* in increasing its congestion window for reducing packet losses and queuing latency, the delivery rate is decreased in all blobs with *newCC*. (3) All blobs show a smaller minimum RTT with *newCC*, indicating that all RPCs have moved from an overloaded, bandwidth-constrained operating point to a more desirable one.

6.4.5 Identifying a Needle in the Haystack

GMM analysis can also reveal subtle but anomalous performance. One of the more distinguishing metrics identified by our GMM analysis was the 5th percentile *cwnd* (Figure 6.14). The “pulse” at the minimum value of blob A raises suspicions and suggests a pathological problem with *newCC*. After sharing this issue with the *newCC* developers, it was traced back to an im-

plementation bug that caused *cwnd* of *newCC* to be stuck at the minimum value in some corner cases. Although these corner cases are rare and have minimal impact on overall performance, our analysis is sensitive enough to highlight them.

6.5 Case Study 4: Infrastructure Upgrade Planning

This case study demonstrates how our GMM-based modeling can help engineers gain insights into the physical constraints that affect the network performance of HDC applications and plan for future infrastructure upgrades. We specifically focus on the performance of *MonitorService*, a global monitoring service responsible for data collection, aggregation, and general monitoring, which is highly sensitive to performance issues. Our analysis is conducted on a heterogeneous datacenter consisting of pods with different switch-port speeds. *MonitorService* is hosted on two types of slow-speed pods, P_{s1} and P_{s2} , as well as one type of fast-speed pod, P_f . For our modeling, we use three days of inter-pod traffic data, which includes 53,278 1-minute t-digest measurements aggregated from over a million sampled RPCs.

Figure 6.15 depicts a 3D scatter plot that shows for the four GMM blobs, the observed 95th percentiles of the three most distinguishing metrics obtained by GMM analysis: *receive queuing latency*, *delivery rate*, and *minimum RTT*. The GMM analysis highlights two types of bottleneck constraints for RPCs from *MonitorService*: network bandwidth and processing on receiver hosts (a detailed analysis is in Section 6.5.1). Our analysis of categorical composition shows that the most distinctive categorical attributes among the different blobs are pod speeds and destination users, as shown in Table 6.8. Blobs 1 and 2 contain RPCs from the low-speed pods P_{s1} and P_{s2} , but are intended for different destination services, d_1 and d_2 , respectively. In contrast, blobs 3 and 4 contain RPCs from the fast-speed pod P_f . RPCs in blob 3 are intended for the same destination service as those in blob 1 (d_1), while those in blob 4 are intended for the same destination service as those in blob 2 (d_2).

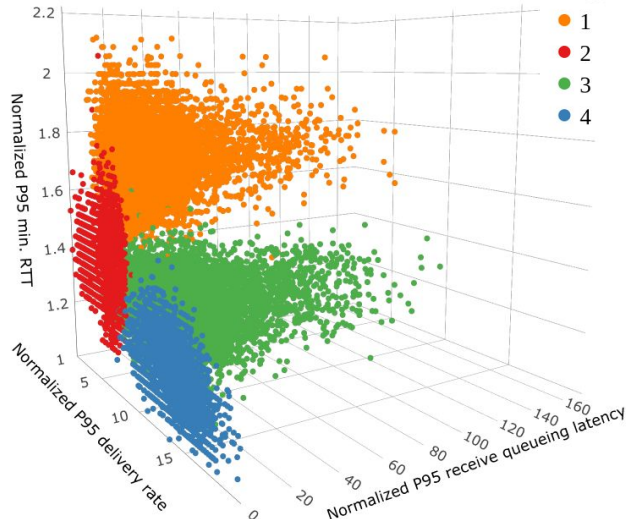


Figure 6.15: *MonitorService*: GMM blobs

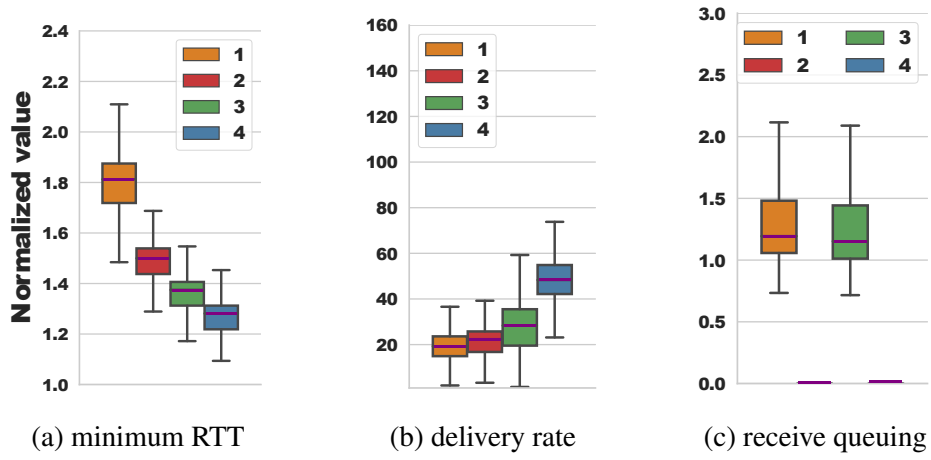


Figure 6.16: Boxplots for the most distinguishing metrics.

<i>Blob</i>	<i>Dst. user</i>	<i>Pod type</i>
1	d_1	P_{s1}, P_{s2}
2	d_2	P_{s1}, P_{s2}
3	d_1	P_f
4	d_2	P_f

Table 6.8: Categorical Composition of Blobs

6.5.1 How Can GMM Inform Planning of Future Upgrades?

Figure 6.16 shows the boxplots of the most distinguishing metrics for RPCs in each blob. We observe that RPCs in the fast-speed pod P_f exhibit higher delivery rates and lower minimum RTT (blobs 3 and 4), which justifies the increased infrastructure and operation cost of upgrading the switch port speeds. However, the results also suggest that the end-to-end RPC latency for user d_1 may not benefit significantly from running on the faster pod P_f . Our GMM-based analysis highlights high receive queuing latency for this user in both slow-speed (blob 1) and fast-speed (blob 3) pods (Figure 6.16(c)), indicating that the bottleneck may be at the receiver and is likely constrained by host resources such as CPU or memory. Therefore, to improve the network performance of d_1 (blobs 1 and 3), upgrading to additional network bandwidth would not be effective. Instead, upgrading the processing resources on the end hosts might be more beneficial. On the other hand, RPCs for d_2 (blobs 2 and 4) could benefit from more bandwidth capacity, as their performance is still restricted by bandwidth even in the fast-speed pod P_f . An alternative approach to boost overall performance without additional hardware upgrades is to decrease the bandwidth allocated to RPCs for d_1 , since the end hosts cannot handle higher workloads. This would allow RPCs in d_2 to use more bandwidth resources to alleviate the bandwidth constraint that limits their performance.

6.5.2 Discussion

Identifying the bottleneck that constrains the performance of HDC applications can be a challenging task. It is often assumed that performance issues are due to the network infrastructure. For example, Guo et al. (19) attribute latencies in applications and kernel stacks to the network. While recent studies have aimed to address this issue, such as Arzani et al. (40) that used a supervised decision tree-based algorithm to identify the responsible entity (client, server, or network) for failures in HDCs, these approaches require large amounts of training data and can not classify previously unseen patterns. The case study demonstrates that by using fine-grained RPC instru-

mentation, GMM can identify non-network bottleneck constraints without any prior knowledge or labelled training data.

6.6 Case Study 5: Root Cause Analysis of Performance Degradation

Diagnosing the root cause of performance issues in HDC applications can be challenging due to the complex interactions between the networking stacks of hosts, application workloads, and underlying storage and network infrastructure. This often requires multiple service owners to collaborate and align the incident timeline with any known changes for identifying the cause. The process can be tedious and time-consuming. In this context, GMM analysis can provide guidance and insights into the root cause of performance issues. The following case study demonstrates how GMM analysis can help with this task.

We conducted an investigation into a significant episode of performance degradation experienced by a key-value store service. The service experienced an unusually high rate of RPC failures in a datacenter, and one potential cause was a recent software upgrade. However, initial analysis of the software did not reveal a clear cause for the failures. The key-value store uses a distributed database service (referred to as *DatabaseService*) that connects multiple storage servers within every pod in the datacenter. *DatabaseService* includes numerous jobs that communicate with a variety of destination services and across different QoS classes. During the incident, it was noted that RPCs from *DatabaseService* to storage servers had generally higher latencies. This led us to consider *DatabaseService* as a potential cause of the performance issues. However, it was also possible that the higher latencies were due to a range of other factors, such as insufficient bandwidth during transmission, exhausted CPU/mem on end hosts, or unfair rate throttling. In short, the underlying cause was not clear from the raw RPC telemetry from Fathom. In fact, in the absence of our analysis pipeline, it took network engineers more than a week to investigate the outage, involving multiple teams, including the application owner, application engineers, network engineers, and the congestion control team. Such a lengthy diagnosis incurred significant costs to the HDC.

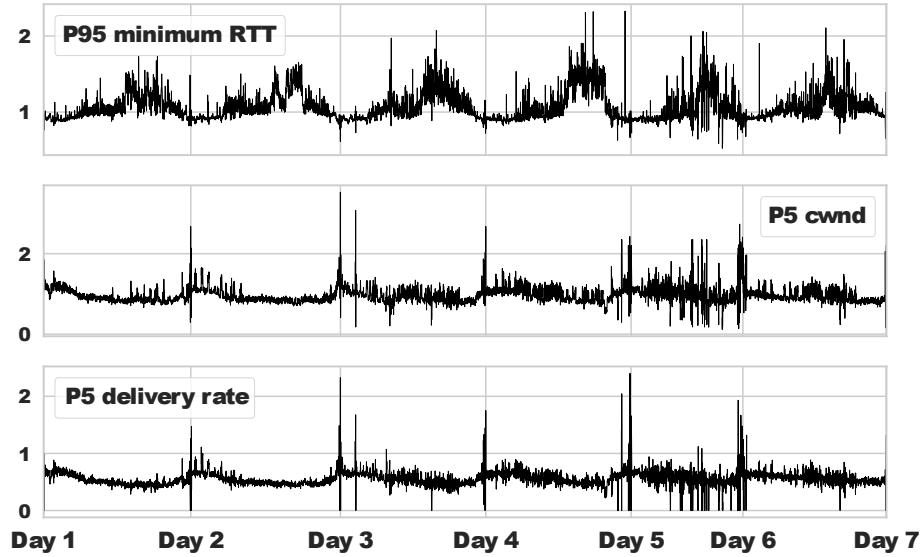


Figure 6.17: *DatabaseService*: Time-series of normalized 95th percentile minimum RTT and 5th percentiles of cwnd and delivery rate across six days.

To illustrate the complexity of the issue, Figure 6.17 shows the time series of the 95th percentile of *minimum RTT* and the 5th percentiles of *cwnd* and *delivery rate* for the two days preceding and the four days following the start of the outage (day 3). The figure reveals numerous spikes with significant fluctuations even before the onset of the outage. It is challenging to distinguish between measurements of “degraded” performance and “normal” spikes based on these time series and predefined thresholds alone. Therefore, identifying the root causes of spikes in multiple metrics would be a daunting task.

6.6.1 How Can GMM Analysis Help Separate Traffic With Degraded Networking Performance?

Our GMM modeling approach conducts multivariate analysis of RPC performance, which considers the joint behavior of multiple performance metrics instead of focusing solely on individual dimensions. This approach has been shown to be more effective in identifying the root cause of degraded performance experienced by application workers, as it can capture the complex interactions between different metrics for understanding the underlying causes of performance issues.
















	P_2	P_6	P_8	P_{25}	P_{26}
P_2	-				
P_3					
P_6		-			

Table 6.9: Traffic volume matrix between top pairs of source (row) and destination (column) pods for measurements in the “best” blob () and the “worst” blob ()

For this case study, we analyzed 172,664 1-minute *Fathom t*-digests measurements aggregated from over 44 million sampled RPCs from *DatabaseService*, collected during the two days following the start of the outage. Our analysis pipeline identified two blobs, one of which had significantly poor network performance. We labelled these blobs as the “best” and “worst” based on their performance. Specifically, RPCs from the “worst” blob had a 95th percentile of *minimum RTT* that was approximately 2.3 times larger, a *delivery rate* that was approximately 5 times smaller, and a *cwnd* that was approximately 15 times smaller. These measurements indicate that their performance is severely constrained by bandwidth resources, and the issue could potentially be resolved by providing additional bandwidth or by distributing the traffic over less congested links. It is worth noting that the GMM-based analysis successfully identified measurements with degraded performance without requiring any prior knowledge about the system. Furthermore, it helped us understand the type of bottleneck and potential remedies.

6.6.2 How Can GMM Analysis Shed Light on the Root Cause?

Our analysis of the categorical composition of the two blobs led to two important observations. Firstly, we found that the jobs most affected in the “worst” blob belonged to an application that was *unrelated* to the key-value store service but was accessing the same set of storage servers through *DatabaseService*. Secondly, we observed that the topological location of the RPCs in the two blobs had a significant impact on their traffic patterns. The traffic pattern of the RPCs in the “best” and “worst” blobs in terms of the source and destination pods is shown in Table 6.9. The diameter of the circles indicates the volume of RPCs between two pods, with larger diameters

indicating a higher volume. RPCs in the “best” blob are more evenly distributed across all pods, while RPCs in the “worst” blob are highly skewed toward communications between pods P_6 and P_2 . The skewed traffic pattern for *DatabaseService* suggested a likely network bottleneck within P_6 or P_2 . These insights guided engineers to identify the following root causes for the performance degradation: (1) The poor performance of the key-value store RPCs was due to congestion experienced by its backend database service (*DatabaseService*). The congestion was caused by an unrelated application that was accessing the same set of storage servers and significantly increased its storage access, resulting in a complete saturation of the bandwidth within the pod. (2) Pods P_6 and P_2 were improperly configured and had insufficient bandwidth for the amount of storage deployed inside. Once the root cause was identified, congestion was easily relieved by doubling the bandwidth capacity within the pods.

6.6.3 Discussion

This case study demonstrates the effectiveness of GMM analysis in extracting valuable insights from complex communication patterns where many interdependent applications share a datacenter network. These applications often share both the software and physical infrastructure, which are constantly upgrading. Performance degradation can be caused by rare or hidden issues that occur a few dependency hops away, making it difficult for inference-based anomaly localization techniques to identify the root cause. These techniques typically rely on detailed dependency graphs to describe the relationships between different components (93; 34; 9). Heuristics-based anomaly detection relies on setting threshold values to identify unusual behavior, while learning-based approaches rely on representative training data to identify patterns. However, both of these approaches can be limited in their ability to generalize to the diverse and large-scale environments found in HDCs. This case study showed that GMM analysis can distinguish between good and poor performance without relying on hand-crafted heuristics, representative training data, or prior knowledge of the infrastructure or workloads. The categorical analysis provided valuable clues about the root cause of the performance issues. While human experts could also eventually

discover this information by slicing and dicing the data, GMM analysis significantly accelerates the process.

6.7 Conclusion

In this chapter, we evaluate the effectiveness of GMMs in modeling network performance of different applications in a production HDC. We use performance metrics collected by Fathom, an existing RPC performance instrumentation, to illustrate the impact of Simpson's paradox on RPC telemetry and demonstrate how GMM analysis can mitigate this issue. Additionally, we present several case studies conducted in the HDC to show how GMM-based analysis can assist in assessment, planning, and troubleshooting tasks for applications running on diverse network infrastructure and over different time periods. In the following chapter, we will discuss the lessons we have learned from conducting further case studies in the production HDC and the improvements we have made to both the performance instrumentation system and our modeling pipeline.

CHAPTER 7: DATA COLLECTION INSTRUMENTATION MODIFICATIONS AND FOLLOW-UP CASE STUDIES

Our GMM analysis for different case studies in the production HDC has revealed two potential issues with Fathom, our performance instrumentation pipeline. Firstly, the modeling results may still be affected by Simpson’s paradox due to the disproportionate ratio of small and large RPCs in the HDC. Secondly, the existing method for collecting the delivery rate in Fathom is not representative of RPC performance due to the volatility of the per-packet delivery rate during the transfer of an RPC.

In this chapter, we discuss these two issues and describe the modifications made to address these. We then present two case studies conducted in the production HDC after the modifications. The first case study demonstrates how GMMs can be used to assess the impact of an engineering change on the performance of a storage service. The second case study showcases how GMMs can be used for fleet-wide analysis when multiple models are built with RPCs from a machine learning service running in different HDC clusters.

7.1 Issue 1: The Impact of RPC Size on Performance

7.1.1 The Network Performance of Small vs. Large RPCs

7.1.1.1 Background: Bandwidth and Delay

Throughput and delay, also known as latency, are two important metrics used to evaluate the performance of a network transfer. Throughput represents the amount of data that can be transferred from the source to the destination within a specific time frame, while delay measures the time taken for a message to travel from one end of the network to the other. Optimal network per-

formance is achieved when the sender can transmit packets at a rate that maximizes throughput and minimizes delay.

Delay is composed of four main components: propagation delays, transmission delays, queueing delays, and processing delays.

1. Propagation delay represents the time it takes for data to propagate between nodes in the forms of an electromagnetic encoded signal. It depends on the distance between two end hosts and the effective speed of electromagnetic wave propagation over the corresponding mediums (as shown in Eq. 7.1).

$$\text{propagation delay} = \frac{\text{distance between hosts}}{\text{propagation speed}} \quad (7.1)$$

2. Transmission delay, also known as serialization delay, is the time it takes to encode the bits of a packet into an electromagnetic signal and transmit it over a physical interface. It is calculated based on the size of the packet and the transmission capacity (link speed) of the interface, as shown in Eq. 7.2;

$$\text{transmission delay} = \frac{\text{packet size}}{\text{link speed}} \quad (7.2)$$

3. Queueing delay measures the amount of time that messages spend waiting in queue at network packet switches before being transmitted on an outbound link.
4. Processing delay occurs at switches and routers along the propagation path over a network. When measuring round trip delay (i.e., the amount of time it takes for a signal to travel from a sender to a receiver and back again), the receiver processing delay is also taken into account.

$$\text{delay} = \text{propagation delay} + \text{transmission delay} + \text{queueing delay} + \text{processing delay} \quad (7.3)$$

$$\text{throughput} = \frac{\text{size}}{\text{delay}} \quad (7.4)$$

7.1.1.2 Background: Physical Constraints Limiting the Performance of Network Transfers

The performance of an RPC can be affected by three main physical constraints along the transmission path: (1) Round-trip propagation delay ($RTprop$), which is the minimum time it takes for a packet to travel from the source to the destination and back as described above. (2) Bottleneck bandwidth ($BtlBw$), which is the transmission capacity of the slowest interface along the path between the source and destination. This value depends on the hardware configurations of the intermediate devices. (3) Bottleneck buffer ($BtlBuf$), which is the available buffer size at the bottleneck switches. The bandwidth-delay product (BDP), which is the product of $BtlBw$ and $RTprop$, represents the maximum amount of data that can be on the network circuit at any given time, that is, data that have been transmitted but not yet acknowledged. For achieving the optimal network performance, the arrival rate of the packets should equal to $BtlBw$ and the total data in flight should equal to BDP (99).

7.1.1.3 What Is Relevant for Small vs. Large RPCs

The performance of RPCs on a network is affected by the three physical constraints discussed above — however, the constraints may have different effects depending on the size of the RPC. Small RPCs are typically less affected by bandwidth, as these are often smaller than BDP and do not fully utilize the available bandwidth on the propagation path; but instead these are impacted by delay, which is mainly determined by $RTprop$. For example, a 1-byte RPC will experience a 100x performance difference with a 100 ms $RTprop$ compared to a 1 ms $RTprop$. Link speed, such as 10 Mbps or 100 Mbps, is relatively insignificant in this case, as the transmission delay would be 0.8 us and 0.08 us, respectively. Therefore, the network performance of small-sized RPCs is most constrained by delay, particularly propagation delay, as their rates and volumes are insufficient to fully utilize the bandwidth resources.

In contrast, large RPCs are mainly affected by bandwidth. For example, it will take 20 seconds to transmit 25 MB data over a 10 Mbps channel, making it relatively unimportant if it is a 1-ms channel or a 100-ms channel. With a fixed propagation delay, a larger bandwidth results in higher throughput. For large RPCs, their network performance is primarily constrained by rates and volumes, as the bottleneck bandwidth determines the amount of data that can be transmitted within a given period and the rate at which it can be sent over the network.

7.1.2 Potential Issues with Disproportional Ratios of Small and Large RPCs

Data center workloads are characterized as heavy-tailed, with a small fraction of RPCs carrying most of the bytes that traverse the network (145; 146; 52). In Google data centers, for example, $\sim 5\%$ of RPCs larger than 64 KB constitute around 87% of the overall traffic. As a result, when Fathom randomly samples a subset of RPCs for monitoring, the majority of them are small. It is important to note that each sampled RPC in Fathom is included in the performance measurement, regardless of its size. When Fathom aggregates the performance metrics of the sampled RPCs within a minute into a t -digest without considering their sizes, it is likely that the overall trends will be heavily influenced by small RPCs due to their abundance, as illustrated by Figures 6.1 and 6.2 in Chapter 6. However, small RPCs do not provide a complete understanding of network bandwidth constraints because (1) their performance is primarily affected by propagation delay and (2) they do not significantly contribute to network load (Section 7.1.1.3). Therefore, Fathom divides *transfer latency* into six buckets based on RPC size: (0, 1KB], (1KB, 8KB], (8KB, 64KB], (64KB, 256KB], (256KB, 2MB], and (2MB, $+\infty$), as shown in Table 6.1 in Section 6.1, to illustrate the impact of sizes on network performance. However, this decision has two potential issues:

1. Distributions along other performance dimensions may still be subject to Simpson's paradox, which can lead to hidden and misleading results. For example, the *delivery rate* of large RPCs may be hidden in aggregated t -digests or only apparent in very high percentiles (e.g., 99.99th percentile). It's worth noting that the *delivery rate* of small RPCs is less infor-

mative in revealing network conditions since they do not fully utilize available bandwidth. On the other hand, high remote latency can occur when a large number of small RPCs burst into the same end host simultaneously, a common workload in machine learning applications when data is merged from multiple monitoring nodes to a central learning agent for training (40). However, this is unlikely to happen with large RPCs, as their performance is mainly constrained by bottleneck bandwidth. As a result, aggregating a mixture of small and large RPCs in each t -digest may lead to Simpson’s paradox, making it difficult to understand their performance. In the above examples, high percentiles of *delivery rate* are determined by large RPCs, while high percentiles of remote latency are determined by small ones. Without further investigation, it is difficult to identify the corresponding percentiles associated with small and large RPCs along each performance dimension.

2. The small number of large RPCs results in a considerable number of aggregated measurements with *empty* t -digests for transfer latency in the large buckets. Specifically, while 99% of the measurements have transfer latency in the 1 KB and 8 KB buckets, less than 5% have at least one valid value in the larger buckets. As mentioned earlier, the transfer latency in each bucket reflects the network experience of RPCs with different sizes. However, machine learning algorithms cannot handle empty values, making it necessary to have at least one sampled RPC in each bucket when using the transfer latency of all six buckets as training features. As a result, the number of RPC measurements available for modeling GMMs is significantly reduced (less than 1%). While including only the transfer latency of small buckets increases the number of measurements, it may not accurately capture the performance of large RPCs, which have a greater impact on the overall network load.

7.1.3 Modifications to Fathom

To address the aforementioned issues, we propose separating sampled RPCs based on their sizes. This results in multiple measurements corresponding to specific sizes, rather than having a single aggregated measurement that includes RPCs of different sizes. However, this separation

will inevitably introduce significant storage overhead in the HDC due to the large volume of performance data. For managing the overhead, we suggest the following changes to Fathom:

1. To classify RPCs based on size, we introduce a categorical dimension with three labels: *small*, *medium*, and *large*. The size range for each label is determined by calculating the BDP based on the minimal physical latency and link capacity in HDCs. In HDCs, most communications occur within a cluster (52), with an average propagation delay of approximately 1 μs . In a generic 3-tier Clos tree topology (28), there are typically 6 to 10 hops between a randomly placed source and destination, with a one-way delay of 0.5 μs per hop for moving data from ingress to egress. Additionally, it takes about 5 μs to handle a typical interrupt on an end host, resulting in a total of 10 μs . Considering the range of NIC speeds in HDCs, which can vary from 10 to 100 Gbps, the BDP is between 21.25 KB (17 μs * 10 Gbps) and 262.5 KB (21 μs * 100 Gbps). RPCs smaller than 21 KB are likely to be dominated by delay, while RPCs larger than 262 KB are likely to be dominated by bandwidth. Based on these two thresholds, we have divided the seven buckets of transfer latency for RPCs of different sizes in Fathom into three categories: *small*, *medium*, and *large*, as shown in Table 7.1. The size label for each sampled RPC is used as an additional categorical attribute for aggregation.

original bucket	size label
(0, 1KB] (1KB, 8KB]	<i>small</i>
(8KB, 64KB] (64KB, 256KB]	<i>medium</i>
(256KB, 2MB] (2MB, inf)	<i>large</i>

Table 7.1: The approach for determining size label for each RPC.

2. To more effectively analyze the impact of sizes on RPC performance, we capture the size of each sampled RPC in Fathom and create a t -digest for tracking its distribution. However, this metric is not used as an input for building GMMs; Rather, it is utilized only for analysis of GMM blobs.

The modifications mentioned above divide a single performance measurement that includes RPCs of different sizes into three separate measurements, each representing the performance of *small*, *medium*, and *large* RPCs. Each measurement contains only one *t*-digest for transfer latency, instead of six, which reduces the storage overhead. Additionally, this new aggregation strategy helps to minimize the impact of Simpson’s paradox on all performance dimensions, as discussed in Section 7.1.2, and enables us to analyze the performance of RPCs of different sizes separately using GMMs.

7.2 Issue 2: Per-RPC vs. Per-Packet Delivery Rate

Delivery rate, also referred to as throughput, is a critical metric for assessing the performance of RPCs in the network, as discussed in Section 7.1.1.1. The original implementation of Fathom only captures TCP’s estimate of the effective *delivery rate* for the last packet in each RPC, which is determined by dividing the size of the last packet by the time gap between the second to last packet and the last packet. However, delivery rate can be fairly volatile when computed on a *per-packet* basis (147). The variability of per-packet delivery rate across packets can significantly impact the accuracy of performance measurements, especially for large RPCs that span multiple packets and are heavily influenced by link capacity. Based on the Fathom data collected in the production HDC, approximately 30% of RPCs transmit more than one packet. Therefore, logging the per-packet delivery rate of the last packet is not representative of the delivery rate experienced by other packets. This limitation is especially relevant for larger RPCs that span multiple packets and are more heavily influenced by link capacity, as their effective delivery rate can vary significantly on a per-packet basis.

To mitigate this issue, we have redefined how Fathom measures *delivery rate*. Instead of relying on TCP’s estimate of per-packet *delivery rate*, we now calculate the average *delivery rate* per RPC by dividing its size by its total transfer latency. Figure 7.1 illustrates the probability distribution of *delivery rate* for RPCs generated by an application in the HDC using both the original and modified approaches. The *per-packet delivery rate* based on the last packet of the connection

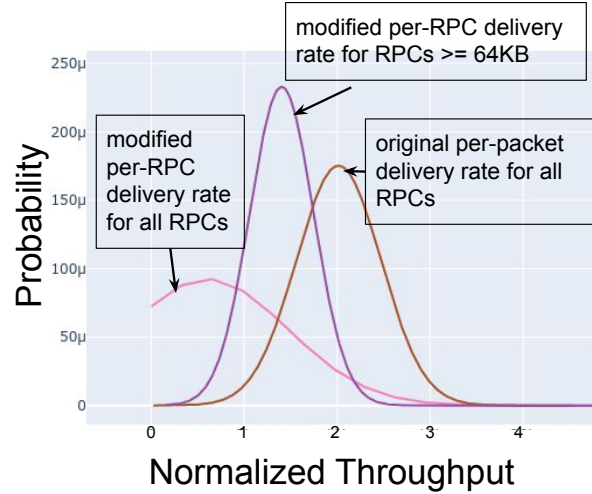


Figure 7.1: The comparison between the original *per-packet delivery rate* and the modified *per-RPC delivery rate* in Fathom.

(original) is about 4 times higher than the *per-RPC delivery rate* calculated from the RPC size and transfer latency (modified). Using the per-packet metric may lead to an overestimation of RPC performance by focusing solely on the rate of a single packet.

We next present two additional case studies conducted using GMMs in the production HDC after implementing the above changes to Fathom.

7.3 Case Study 1: Assess the Impact of Protective Load Balancing (PLB) Deployment

PLB (56) is designed for load balancing traffic in data centers by rerouting congested or disconnected flows based on end-to-end congestion notifications (ECN (65)) and failure signals (TCP retransmission timeouts). Previous studies have demonstrated the effectiveness of PLB in reducing the average and tail latencies of all-to-all workloads compared to state-of-the-art techniques through simulations and testbed evaluations (148). In this case study, we aim to assess the impact of deploying PLB at a larger scale in production HDCs on the network performance of a storage service.

Storage traffic accounts for the majority of data center traffic at Google, with many applications relying on RPCs over TCP for read and write operations. However, workload changes

can quickly create hotspots on ToR switches hosting these storage servers. For assessing the impact of PLB on the storage service, we built two GMMs to model the network performance of the RPCs of the storage service in a representative data center cluster. We analyzed the RPCs for a period of 7 days before and after the PLB deployment, excluding the rollout time window. During this period, we did not observe any significant change in the overall workload.

We plotted the time series of *delivery rate* and *transfer latency* before and after the deployment in Figure 7.2, which showed spiky patterns, as expected, with no clear trends. However, by comparing the GMMs, we identified two groups of RPCs (blobs A and B) whose performance was most affected by PLB — the network performance of the remaining 2 blobs was not affected much by the PLB deployment. These groups have distinct categorical attributes, revealing important insights into how PLB impacted the storage service’s network performance.

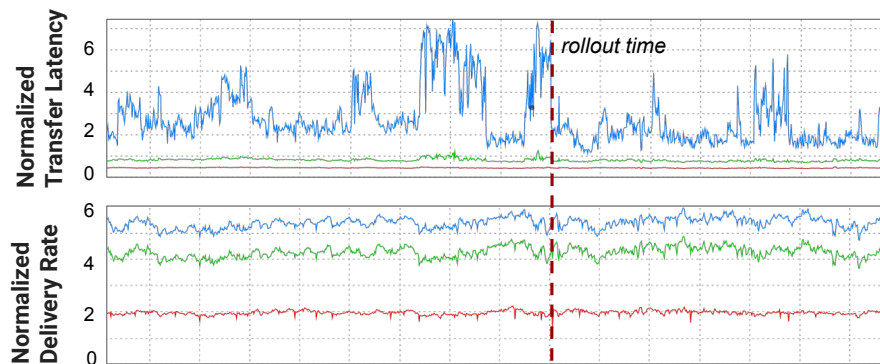


Figure 7.2: Time series of the normalized 99th percentile (blue), 95th percentile (green), and 50th percentile (red) of transfer latency and delivery rate of the storage service RPCs before and after the deployment of PLB.

RPCs in blob A are related to a specific job and have small sizes, with a maximum of 1 KB, and low QoS requirements. These small RPCs often serve as control messages for the storage system and need to be delivered quickly, and their latency is lower-bounded by the physical propagation delay, which PLB cannot change. Figure 7.3 shows the tail latency of these RPCs decreased after the deployment of PLB. Specifically, the 99th percentile has dropped by about 20%, while the first 25th percentile has somewhat increased. The increase in the first 25th percentile can be attributed to more similar levels of congestion experienced by all small RPCs after the

deployment of PLB. Before PLB, some of these transfers experienced much heavier congestion due to load imbalance, resulting in a more variable distribution of latencies.

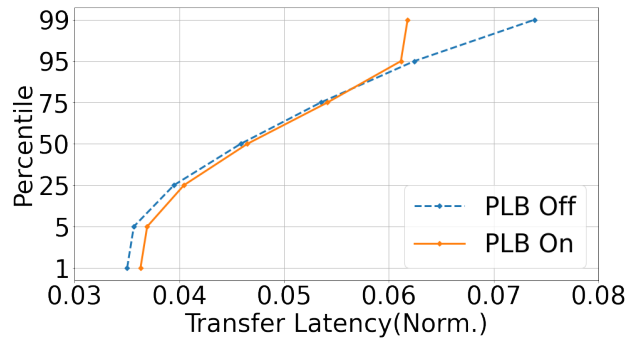


Figure 7.3: Latency for small ($\leq 1\text{KB}$) RPCs of storage workload.

RPCs in blob *B* are related to a different job and have sizes larger than 2 MB, with a higher QoS requirement than the RPCs in blob *A*. These RPCs carry storage chunks, and their latency is reduced across all percentiles, as shown in Figure 7.4. The reduction starts at around 5% and increases to 10% at the 99th percentile. This improvement is due to the fact that as PLB resolves collisions by spreading heavy flows, it finds more available bandwidth and increases the effective capacity for all flows.

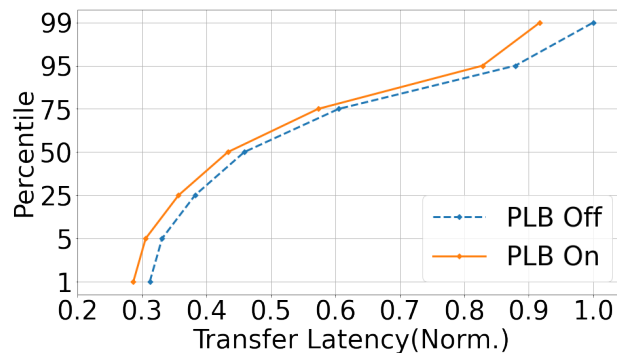


Figure 7.4: Latency for large (2MB+) RPCs of storage workload.

In this case study, GMM highlights two subsets of RPCs whose behaviors are impacted by PLB in different ways. With traditional approaches, it is challenging to identify the subset of RPCs that are being affected and learn their changing trends as shown in Figure 7.2.

7.4 Case Study 2: Evaluate the Fleetwide Performance of Machine Learning Service in HDCs

As machine learning applications become increasingly prevalent in our daily lives, the volume of machine learning workloads also increases in data centers that support such applications. In this case study, we analyze the performance of RPCs from a representative distributed machine learning service, referred to as MLService, in the production HDC. Unlike our previous case studies, which focused mainly on the performance of multiple applications running in one HDC cluster, this case study evaluates the performance of MLService in eight HDC clusters over the course of a week. These clusters account for more than 60% of the total MLService traffic.

During the week, there were over 10 million sampled RPCs from MLService in each cluster, which were aggregated into over 100,000 RPC measurements. The HDC clusters are located in different geographical locations with diverse infrastructures, including machine types, NIC speeds, and OS versions, and host different applications (in addition to MLService) simultaneously. To help us understand the performance differences experienced by RPCs across these clusters, we gather external categorical information describing the configuration of each machine in each cluster, including storage configuration (such as the number of SSDs and disks), computation configuration (such as the number of CPUs and cores), and network configuration (such as the number of NICs and NIC speed). The results indicate that machines have different configurations across these clusters in all three aspects. By considering this information alongside our performance analysis, we can gain insights into the factors that affect the performance of the MLService RPCs in these clusters.

In this case study, we faced two new challenges when analyzing GMMs results of RPCs in multiple clusters and developed new techniques to address them. In the following sections, we discuss these two issues and their solutions. Finally, we summarize our findings about the performance of MLService using GMMs.

7.4.1 Issue 1: Job Name Analysis

The source and destination job of each RPC is an important categorical attribute that characterizes RPCs in different blobs in GMM analysis. As different jobs can represent different workloads and communication patterns that lead to different performance, it is important to determine if RPCs in a GMM blob belong to the same job or if RPCs in different GMM blobs are from different jobs. This information can help us understand why RPCs perform in certain ways and highlight which jobs are most affected by performance anomalies, as discussed in Chapter 6.

However, the job names defined in MLService are too detailed, making it challenging to identify common or distinct job information in the GMM analysis. The general format of a job name in MLService is “< *field_1* > . < *field_2* > . < *field_3* > . < *field_4* >”, where each field contains multiple substrings composed of both letters and integers concatenated by a hyphen (“-”). When using job names to calculate cosine similarity and entropy in GMM analysis, it is difficult to distinguish between blobs with RPCs that differ only in the last field of their job names and those that are completely different, as their cosine similarity and entropy are the same.

To extract high-level job information for analysis, we use TF-IDF (Time Frequency-Inverse Document Frequency) (57) to measure the importance of each substring in job names. TF-IDF compares the number of times a word appears in a document with the number of documents in which the word appears, as formalized in Equation 7.5.

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t) = \frac{f_{t,d}}{\sum_{t'} f_{t',d}} \times \log \frac{N}{|\{d \in D : t \in d\}|} \quad (7.5)$$

where t is the term (word) we are looking to measure the commonness of and N is the number of documents (d) in the corpus (D).

In our case, we tokenize the source and destination job names by using “.” and “-” as delimiters. Each substring is considered a word, and each source/destination job name of an RPC is considered as a document. After calculating the weight of each word using TF-IDF, we select the important words to regenerate the source/destination job name of each RPC. For instance, given

an original job name “*flow-tree-main-f209b281.worker-poolf-v8-data.eval-043-combine.grass-server-10d*”, we generate a new name “*flow-tree.worker.eval-combine.grass-server*” by retaining only the important words based on TF-IDF weights.

These newly generated job names are used for calculating cosine similarity and entropy, which helps to better reveal similarities and differences in job names for RPCs between blobs. For example, Table 7.2 shows the *average* cosine similarity of original and processed source job names between five blobs in different GMMs. The results demonstrate that after processing the job names using our approach, it becomes evident that the RPCs in blob 5 are from different jobs compared to others.

blob ID	1	2	3	4	5
Original source job name	0.19	0.18	0.21	0.12	0.18
Processed source job name	0.6	0.72	0.75	0.63	0.18

Table 7.2: Cosine similarity between job names that appear in five blobs in different GMMs based on the original and processed job names.

7.4.2 Issue 2: Match GMM blobs in Different DataCenter Clusters

When comparing multiple blobs from GMMs, such as before and after a rollout, we use the Wasserstein distance and categorical attributes to match blobs and ensure an “apples-to-apples” comparison. This matching approach has been effective in our previous case studies, both in CloudLab (Chapter 5) and in the production HDC (Chapter 6), when studying the performance of RPCs from one or more applications within the same HDC cluster over different time periods.

However, due to infrastructure differences across HDC clusters, the actual performance of RPCs can vary significantly, even when they are limited by the same type of constraint. This variability can make the current matching condition too strict, making it challenging to find corresponding blobs across all GMMs and accurately compare RPC performance in different clusters. For example, Figure 7.5 shows the 2D distribution of RPCs from MLService in three HDC clusters in terms of *delivery rate* (x-axis) and transfer latency (y-axis) based on GMMs. The plots show that although the relative locations of the blobs in these three GMMs are similar, RPCs

in blob 1 experience a higher *delivery rate* in cluster C. GMM analysis revealed that cluster C has more network resources with a larger number of NICs and higher NIC speeds compared to clusters A and B, which explains the gap in delivery rate. However, when matching blobs using our previous approach, only two blob pairs can be exactly matched across these three GMMs, as shown in Table 7.3. For the remaining unmatched blobs, it is difficult to understand how their performance differs from or resembles others. Therefore, we have relaxed the matching criteria to account for significant infrastructure differences between HDC clusters, which can cause large variations in RPC performance despite similar constraints.



Figure 7.5: 2D distribution of each GMM blob for RPCs in MLService from two three clusters (x-axis: normalized delivery rate, y-axis: normalized transfer latency).

	A	B	C
matched pair 1	1	1	1
matched pair 2	2	3	3

Table 7.3: Blobs matched in the three GMMs based on the Wasserstein distance.

To compare the performance of different clusters and identify similarities and differences between them, we use hierarchical agglomerative clustering (58) based on the Wasserstein distance. This approach allowed us to build a hierarchy of GMM blobs that indicates how different or similar their performance is. By analyzing the dendrogram in Figure 7.6, we could see the hierarchical relationship between GMM blobs, with the height indicating the distance between the blobs.

One of the advantages of this approach is that it not only enables us to match blobs, but also provides information about the performance differences of unmatched blobs. Specifically, we are

able to obtain the same information as in Table 7.3 from the dendrogram, since the matched blobs in Table 7.3 ($C : 1, A : 1$ and $B : 1$; $A : 2, B : 3$ and $C : 3$) have the smallest distance between each other, as shown in Figure 7.5. Additionally, from Figure 7.5, we can learn the following:

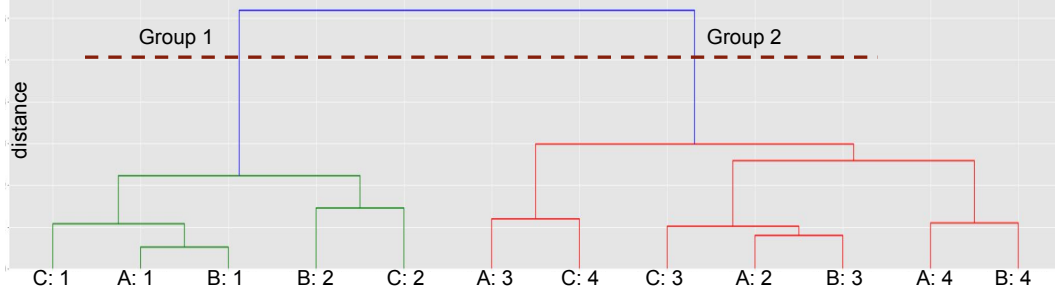


Figure 7.6: Hierarchical clustering dendrogram of the GMM blobs in Figure 7.5.

1. The GMM blobs can be divided into two distinct groups based on the distance, Group 1 and Group 2. The GMM analysis reveals that the blobs in Group 1 ($C : 1, A : 1, B : 1, B : 2$, and $C : 2$) contain RPCs from jobs with larger sizes, while the blobs in Group 2 ($A : 3, C : 4, C : 3, A : 2, B : 3, A : 4$, and $B : 4$) include RPCs from other jobs with smaller sizes.
2. The unmatched blob $B : 2$ and $C : 2$ have similar performance and are closer to blobs $C : 1, A : 1$, and $B : 1$. The GMM analysis shows that the RPCs in these two blobs differ the most from other blobs in Group 1 in terms of smaller congestion window size, suggesting that their performance is more likely to be constrained by volume.
3. The RPCs in the remaining four unmatched blobs have a lower delivery rate, compared to the previously matched blobs in Group 2 ($A : 2, B : 3$, and $C : 3$). Among these blobs, $A : 4$ and $B : 4$ have similar performance and are more different from $A : 3$ and $C : 4$. GMM analysis shows that compared to RPCs in $A : 3$ and $C : 4$, RPCs in $A : 4$ and $B : 4$ have a smaller receive queueing latency, indicating that their performance is less constrained by remote hosts.

The above example illustrates that this clustering approach for GMM analysis can significantly simplify the analysis process when dealing with a large number of blobs. By grouping

similar blobs together, we can efficiently extract critical information and insights, reducing the time and effort required to analyze the data.

7.4.3 What Can Be Learned from the Fleet-Wide Analysis

Through the analysis of the GMM blobs of MLService from the eight HDC clusters, we have demonstrated that the relative performance of the blobs in each cluster is generally consistent, with minor variations due to workload and infrastructure differences, as shown in Figure 7.5. The scalability of HDCs and the number of applications running inside make generic models critical for performance monitoring, as it would be overwhelming to maintain unique models for RPCs from each application running at each location. Furthermore, if that were the case, it would indicate that our initial assumption about using GMMs as the appropriate physical model for describing the performance of RPCs was incorrect.

Using the new techniques described earlier, GMM analysis can efficiently identify blobs with similar or different performance, indicate the most distinct/uniform performance metrics between blobs, and highlight the most distinguishing categorical attributes corresponding to these blobs when there are tens of GMM blobs. By analyzing GMM blobs from these eight HDC clusters, we can learn about the network performance of RPCs in each blob and their constraints by identifying the dominant latency. Specifically, our analysis of GMM blobs from the eight HDC clusters has provided the following insights:

1. In the case of large RPCs, all blobs are network-constrained. However, there is one blob where RPCs experience significantly higher transfer latency in the network, as illustrated in Figures 7.7(a) and 7.8(b). GMM analysis reveals that RPCs in this blob belong to a unique set of jobs. Table 7.4 indicates that the average cosine similarity in source jobs between two other blobs in the HDC cluster is around 0.93, while it is only 0.04/0.17 between blob 1* (which has abnormal performance) and blob 2/3. Moreover, the average size of RPCs in this blob is four times larger than that of RPCs in other blobs. Machine configurations show that machines in this cluster have the smallest number of NICs per machine and the

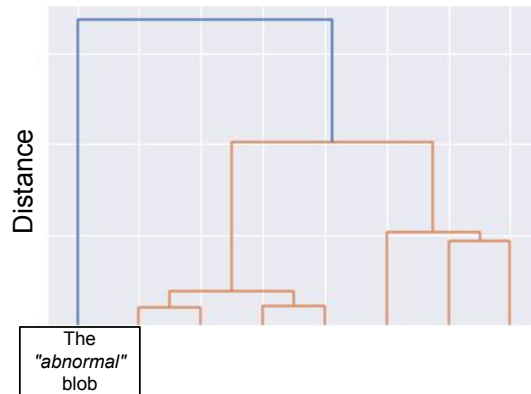
slowest NIC speed. As a result, larger RPCs are more constrained when there are fewer bandwidth resources available in this cluster. To improve the performance of these RPCs, one option is to place them in other clusters with more bandwidth resources.

<i>blob pair</i>	(1*,2)	(1*,3)	(2, 3)
<i>cosine similarity</i>	0.17	0.04	0.93

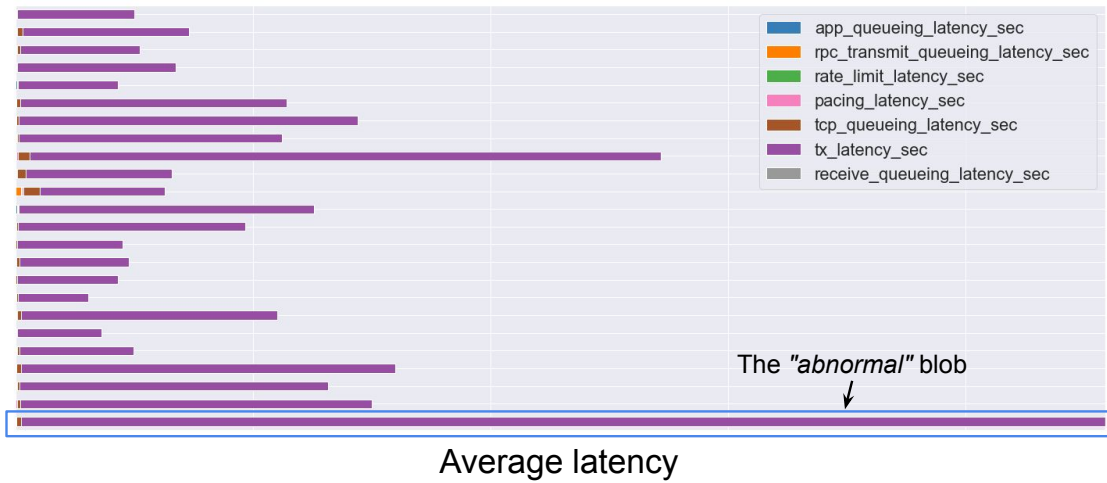
Table 7.4: Cosine similarity of source jobs between GMM blobs in the HDC cluster.

2. For medium-sized RPCs, the network is the primary constraint for all blobs except one, where the workload exhibits different characteristics based on GMM analysis. In particular, despite having a comparable overall traffic volume to other blobs, this blob has nearly four times more RPCs sent in each aggregated interval, while the average RPC size is approximately four times smaller. This leads to a significant local delay that is almost 10 times larger than in other blobs. GMM analysis, therefore, suggests the RPCs in this blob into larger ones could reduce local latency and enhance overall performance.
3. In 7 out of 8 HDC clusters, small RPCs in MLService are mainly constrained by receivers, resulting in a dominant queueing delay on the remote host. However, in one blob in the remaining HDC cluster, the performance of small RPCs in one blob is constrained by senders, as shown by a slightly increased local delay (about two times) compared to RPCs in other clusters, as shown in Figure 7.9. GMM analysis shows that the jobs in this blob are unique, as they are accelerated by Tensor Processing Units (TPUs) based on their job names. This reduces the queueing delay on remote hosts by up to 80% while sending nearly four times more RPCs in a fixed interval.

In summary, this case study has demonstrated that the performance of RPCs within the same application is generally consistent across multiple HDC clusters, despite differences in infrastructure and workload. This finding suggests that the results obtained through GMM can be used for large-scale global HDC performance monitoring. Moreover, GMM analysis can identify sub-

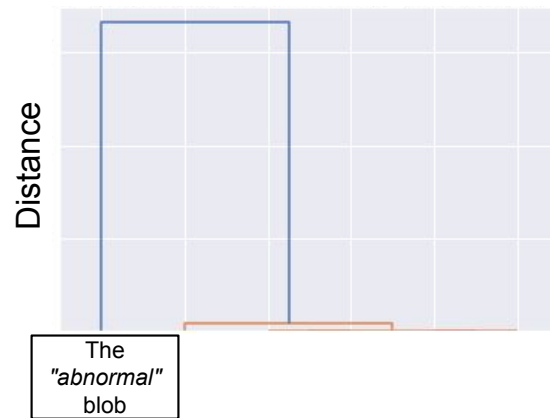


(a) Dendrogram of GMM blobs based on hierarchical clustering results.

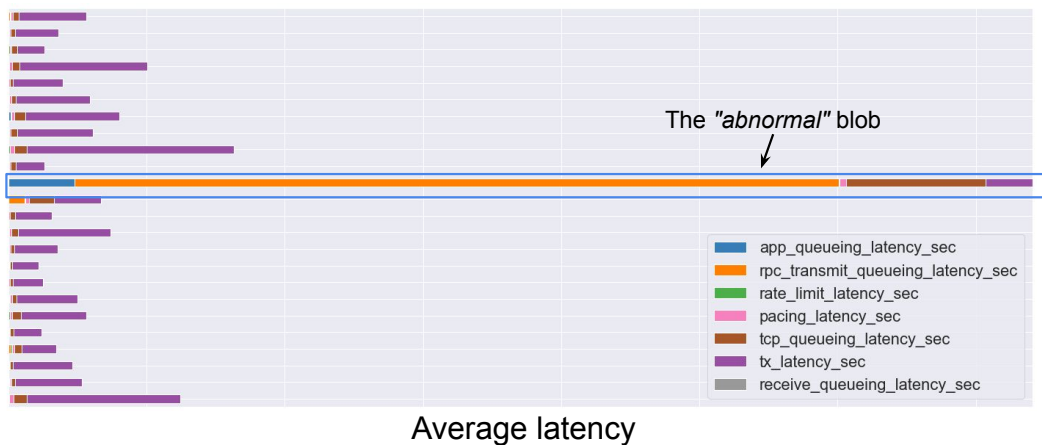


(b) The breakdown of average latency in each GMM blob.

Figure 7.7: Distance between GMM blobs and the breakdown of average latency in each blob for large RPCs.

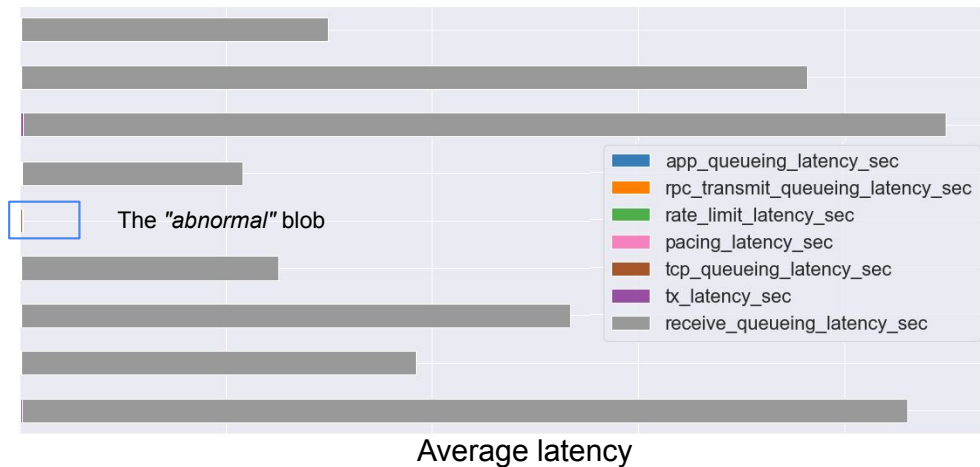


(a) Dendrogram of GMM blobs based on hierarchical clustering results.

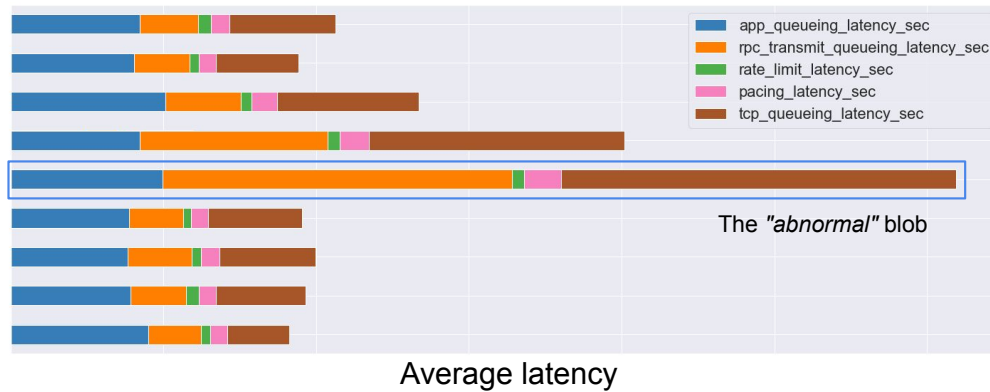


(b) The breakdown of average latency in each GMM blob.

Figure 7.8: Distance between GMM blobs and the breakdown of average latency in each blob for medium-sized RPCs.



(a) Overall latency.



(b) Local latency.

Figure 7.9: The breakdown of average latency in each GMM blob for small-sized RPCs.

sets of RPCs that experience different performance, and highlight the most distinct performance metrics and categorical attributes, providing valuable insights into their performance.

7.5 Conclusion

In this chapter, we discuss two issues in our performance instrumentation pipeline in the production HDCs that affect the accuracy of the modeling results, and describe our approaches for addressing these. The modified pipeline is then used for two additional case studies that demonstrate the effectiveness of GMM-analysis in (1) assessing the impact of a new load-balancing policy on application performance, and (2) evaluating the fleetwide performance of a machine learning service in the production HDCs. For extracting useful insights from the GMM results in the fleetwide analysis, we introduce two techniques to the GMM analysis component. The first one is TF-IDF, which helps to identify distinct categorical attributes among GMM blobs. The second technique is hierarchical agglomerative clustering, which enables us to group GMM blobs with similar performance. These two techniques make it easier to analyze and interpret the GMM results.

CHAPTER 8: CONCLUSIONS AND FUTURE WORK

8.1 Conclusions

Understanding the network performance of applications in HDCs is essential for assessing their overall performance, planning for future upgrades and rollouts, and troubleshooting performance anomalies. However, due to the massive scale, diverse infrastructure and workload, and inherent coupling due to shared resources in HDCs, this task is inherently challenging. In this thesis, we propose the use of constraint-based models for modeling the network performance of HDC applications. We have implemented a pipeline that takes RPC telemetry as input, models different RPC behaviors with GMMs, and analyzes the modeling results using a set of statistical tools.

Over the past three years, we have conducted several case studies on a controlled emulation testbed and in a production HDC, demonstrating the effectiveness of our constraint-based modeling approach in (1) identifying RPCs experiencing different network performance and their respective bottleneck constraints, (2) assessing the impact of engineering changes and infrastructure differences on application network performance, and (3) detecting and troubleshooting performance anomalies.

8.2 Role of Constraint-based Modeling in Achieving Different Network Health Monitoring Goals

Network monitoring in data centers has been a fairly active research field in the past decade. There have been significant advances made in both anomaly detection (identifying traffic experiencing anomalous performance (46; 41; 47; 45)) and anomaly classification (determining the

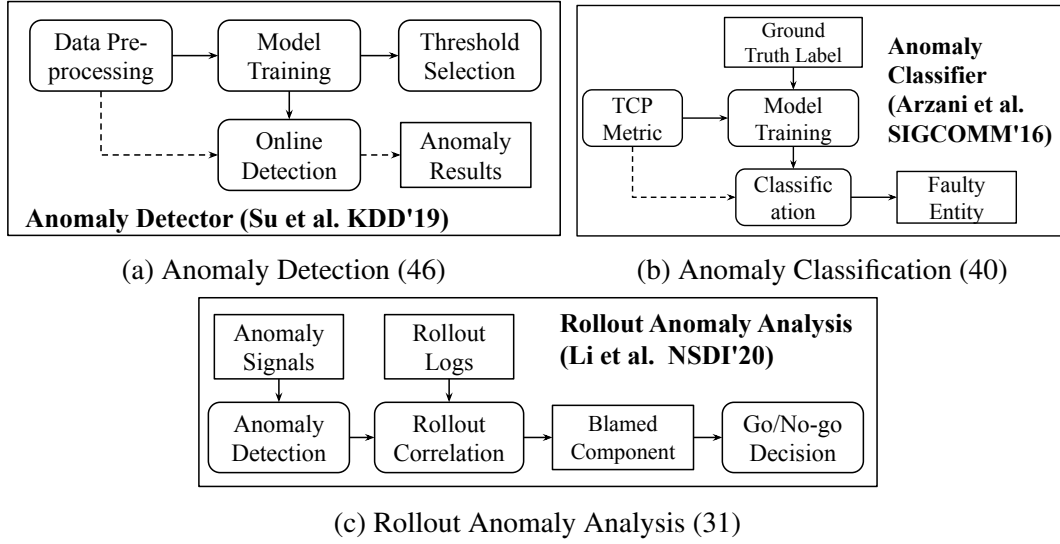


Figure 8.1: Analysis Pipelines Adopted in Key Prior Work

entities responsible for anomalies, including during rollouts (40; 35)). Figure 8.1 illustrates the respective pipelines used in prominent recent approaches.

The main focus of prior work has been on addressing performance pathologies, by designing anomaly detection and classification approaches. As noted in Section 1.1.1, however, HDC network operators have several important goals with respect to network monitoring—in addition to anomaly detection and classification, they would significantly benefit from developing a general *sense* of the network performance of applications, for purposes of long and short-term planning, understanding the impact of infrastructural and engineering changes, as well as understanding the root cause and fix of performance pathologies. Our constraint-based analysis approach provides a scalable and interpretable lens for such *sense-making* analysis, that focuses not only on the bad, but also on the good, and the moderate—and in fact, naturally separates these out from each other.

Figure 8.2 envisions a general framework for network monitoring in HDCs, in which GMM-based constraints analysis helps inform all of the different network monitoring goals mentioned above. Specifically, the ability of GMM to distill out distinct categories of network performance, combined with our contributions in (i) characterizing the performance of different GMM blobs, (ii) categorical composition analysis, and (iii) comparison of blobs before and after a change, can

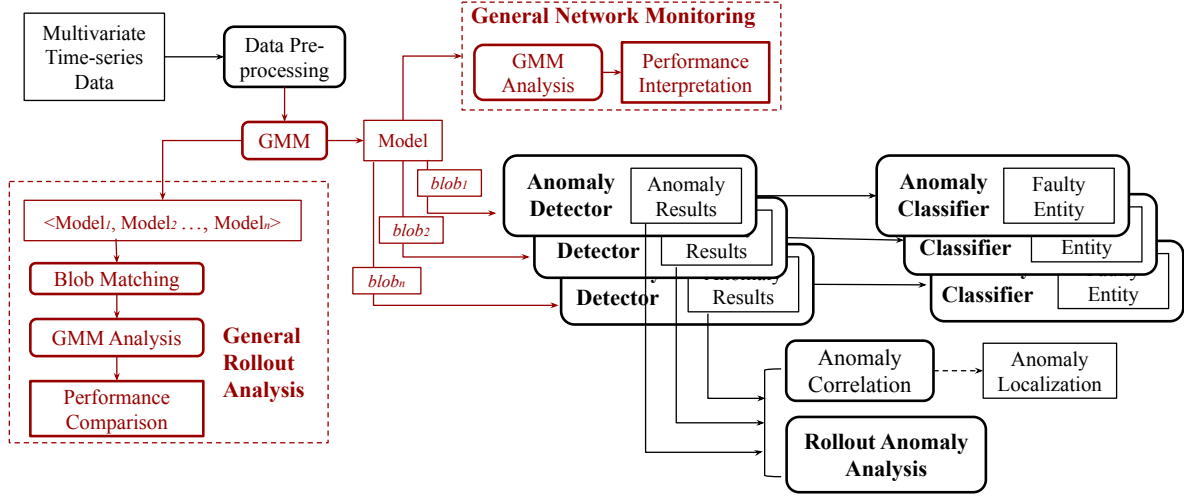


Figure 8.2: A General Network Monitoring Framework—GMM analysis informing network health monitoring and planning, general rollout impact analysis, anomaly detection, and anomaly classification.

be a significant enabler with several applications. First and foremost, we have already demonstrated our ability to empower network operators with sense-making analysis for network health monitoring and planning, as well as understanding the impact of infrastructural change—to the best of our knowledge, this has never been achieved before. Second, Figure 8.2 envisions that our approach can also benefit previously-proposed approaches for anomaly detection/classification—by first distilling performance data based on distinct network performance behavior, and then applying these approaches to each blob. The importance and need for doing so is also supported by observations in (40; 35): Due to the significant diversity in implementation, service types, and traffic patterns of HDC applications, learning-based anomaly detection and classification may need to individually train different detectors/classifiers for each application. In this case, GMM can help automatically identify application instances that experience the same network performance—thus, only one detector/classifier is needed for them (Figure 8.2). Consequently, this can help achieve significant gains in classification/detection performance, without incurring the overhead of labeling a large volume of training data as well as building thousands of models on a per-application basis.

8.3 Limitations and Future Work

Our research has developed a scalable interpretable approach for understanding the network performance of HDC applications. However, the interpretation of our modeling results still requires the expertise of domain experts in HDCs. Although we can group RPCs experiencing different behavior into different blobs and highlight the most distinct performance metrics and categorical attributes associated with each blob in a GMM, it still requires additional effort to understand how the observed network performance and the categorical attributes are connected for revealing the underlying determinant factors.

Our ultimate vision is to develop an automated analysis framework that leverages our modeling approach to summarize the network performance of different HDC applications, to assess the impact of planned infrastructure and engineering changes, as well as to understand the root causes behind unplanned performance outages. Such a framework would significantly reduce the time and workforce required for analyzing the network performance of different applications in HDCs, compared to the current state of the art, which only highlights the most distinguishing performance metrics and categorical attributes in GMMs.

We believe that the scalability, generalizability, and strength of our approach can be further enhanced by pursuing the following directions:

1. Exploring new tools and techniques for interpreting GMM modeling results. These can help reveal more important information associated with each GMM blob regarding their network behaviors. A promising avenue for further exploration is to leverage the correlations between different performance metrics in GMMs. These correlations can provide valuable insights into the bottleneck constraint that limits the network performance of RPCs in each GMM blob, as described in Chapter 3. For example, if high network latency is correlated with low throughput for a particular GMM blob, it suggests that network bandwidth is a bottleneck constraint that limits the performance of RPCs in that blob. By investigating these correlations in more detail, we can gain a more nuanced understanding of the

performance behaviors identified by GMMs and how to improve network performance for HDC applications. To fully leverage these correlations, however, we need to integrate new tools and techniques for summarizing this information in the analysis pipeline.

2. Developing approaches for summarizing GMM modeling results at scale. Fleetwide analysis helps to understand how different factors, such as diverse network infrastructure, competing traffic, and geographic locations, affect application performance by comparing GMMs built from RPCs across multiple clusters. As HDC infrastructure and workload continue to grow in size and complexity, new techniques are required to efficiently perform large-scale fleetwide analysis, as demonstrated in Chapter 7. Therefore, we must continue to explore additional tools and techniques for addressing potential issues in fleetwide analysis.
3. Integrating domain knowledge to label performance behavior characterizations identified by GMMs. When a GMM is trained, it can identify different performance behaviors based on the performance data it receives. However, interpreting these behaviors can be challenging without additional context and knowledge. By incorporating domain knowledge, experts can provide meaningful labels for each GMM blob, such as network-constrained, application-constrained, and sender/receiver-constrained, that accurately reflect the performance behavior characterizations identified by GMMs. Combined with categorical attributes, such as traffic classes, job names, or geographic locations, these labels can be useful for assessment, planning and troubleshooting tasks, as demonstrated in Chapters 5 - 7.

This labeling process creates labeled data that can serve as ground-truth training data for building supervised learning models. These models are a crucial step toward automating GMM analysis and reducing manual analysis overhead, which allows us to understand application network performance more efficiently.

4. Improving monitoring instrumentation with higher accuracy and precision. Monitoring instrumentation plays a crucial role in collecting data for GMM modeling. The quality of input data collected by the underlying monitoring instrumentation directly affects the accuracy of modeling results. Therefore, it is essential to improve the accuracy and precision of the data collected by the monitoring instrumentation, as demonstrated in Chapter 7. There are several ways to achieve this goal. For example, (64) proposed a learning-guided sampling method for collecting more data related to performance outliers, which occur less frequently than normal performance and are represented less frequently in the monitoring data. This approach can ensure that data related to infrequent events is collected, which is important for root causing. By continuing to improve the accuracy and precision of the input data collected by the monitoring instrumentation, we can ensure that the GMM modeling results are more reliable and accurate, which can help us make better decisions regarding the HDC infrastructure and application performance.
5. Incorporating hardware-level and application-level performance metrics into the data collection pipeline. Although this thesis primarily focuses on application network performance, we believe that our approach can be extended to other subsystems, such as storage, computation, and memory, by incorporating the corresponding performance metrics. Collecting these additional performance metrics can be challenging, as it requires additional instrumentation at the hardware and application levels. However, with advances in monitoring tools and technologies, it is becoming increasingly feasible to collect these metrics in a scalable and efficient manner. By incorporating these metrics into the data collection pipeline, we can develop more accurate and comprehensive models that can provide valuable insights into the performance of different subsystems and their impact on the overall system performance.

APPENDIX A: MONITORING TOOLS ON CLOUDLAB

A.1 SNMP

SNMP is a networking monitoring protocol that is used to collect information on devices connected to the network (53). Table A.1 shows the detailed information collected through SNMP. During experiments, we query the above information for each switch based on the corresponding OIDs every second.

Field	OID	Description
ifHCInOctets	1.3.6.1.2.1.31.1.1.1.6	Incoming bytes
ifHCOctets	1.3.6.1.2.1.31.1.1.1.10	Outgoing bytes
ifInUcastPkts	1.3.6.1.2.1.2.2.1.11	Incoming No. of packets
ifOutUcastPkts	1.3.6.1.2.1.2.2.1.17	Outgoing No. of packets
ifInDiscards	1.3.6.1.2.1.2.2.1.13	Incoming No. of discards
ifOutDiscards	1.3.6.1.2.1.2.2.1.19	Outgoing No. of discards
ifCounterDiscontinuityTime	1.3.6.1.2.1.31.1.1.1.19	Indication of discontinuities of interfaces' counters

Table A.1: Information collected via SNMP on switches.

A.2 Ifconfig

Ifconfig is a system administration utility in Unix-like operating systems for network interface configuration. It returns information about all network interfaces currently in operation on the machine, including the total number of bytes and packets transmitted (tx) and received (rx) over each interface. Figure A.1 shows an example output of **ifconfig**. During our experiments, we run **ifconfig** on all end machines on the testbed every second.

```
eno50: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
inet 192.168.1.15 netmask 255.255.255.0 broadcast 192.168.1.255
inet6 fe80::9af2:b3ff:feca:d0c1 prefixlen 64 scopeid 0x20<link>
ether 98:f2:b3:ca:d0:c1 txqueuelen 1000 (Ethernet)
RX packets 7186138 bytes 577780212 (577.7 MB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 10902521 bytes 16432974969 (16.4 GB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figure A.1: Example output of **ifconfig**.

	frame_time_relative	ip_src	tcp_srcport	ip_dst	tcp_dstport	tcp_seq	tcp_len	tcp_acks_frame	tcp_flags_str
frame_number									
5601411	57.253194	192.168.1.77	33160	192.168.1.13	80	0	0	NaN	SYN
5601524	57.253768	192.168.1.13	80	192.168.1.77	33160	0	0	5601411.0	ACK+SYN
5601525	57.253775	192.168.1.77	33160	192.168.1.13	80	1	0	5601524.0	ACK
5601529	57.253790	192.168.1.77	33160	192.168.1.13	80	1	160	NaN	ACK+PSH
5601567	57.254406	192.168.1.13	80	192.168.1.77	33160	1	0	5601529.0	ACK
5601571	57.254487	192.168.1.13	80	192.168.1.77	33160	1	1299	NaN	ACK+PSH+FIN
5601572	57.254501	192.168.1.77	33160	192.168.1.13	80	161	0	5601571.0	ACK
5601573	57.254529	192.168.1.77	33160	192.168.1.13	80	161	0	NaN	ACK+FIN
5601575	57.255211	192.168.1.13	80	192.168.1.77	33160	1301	0	5601573.0	ACK

Figure A.2: An example trace captured by tcpdump.

A.3 Tcpdump

Tcpdump is a command-line utility that captures and analyzes network traffic that passes through the system (55). During experiments, tcpdump is launched to capture information about TCP/IP and packets that are transmitted and received over a network on both servers and clients. This information is stored in pcap files and analyzed using *wireshark*. Figure A.2 shows an example trace, from which we can identify connection establishment via a three-way handshake (that is, 5601411-5601525), data transfer (that is, 5601529-5601572: client sends request and server sends response), and connection termination (that is, 5601573-5601575).

APPENDIX B: ADDITIONAL CASE STUDIES ON CLOUDLAB TESTBED

B.1 Experiment 1: The Impact of HTTP Keep-Alive with a Single Connection

In the first experiment, we evaluate the effect of HTTP Keep-Alive on RPC performance when there is at most one connection between each server/client pair. The experiment involves the transfer of approximately 1 GB of data from the server to the client through multiple HTTP requests and responses. A new request is sent shortly after receiving the full response to the previous request and the response size follows the flow size distribution of the cache application described in Section 5.1.1. To run the experiment, we use the *wget* command with or without *-no-http-keep-alive* flag to enable or disable HTTP Keep-Alive, respectively.

What Can Be Learned from Traditional Analysis Table B.1 presents a comparison of the average performance in terms of throughput, the 95th percentile of bytes-in-flight, and RTT. The results indicate that enabling HTTP Keep-Alive leads to improved network performance for RPCs, with a higher throughput (114%), a higher 95th percentile of bytes-in-flight (131%), and a smaller 95th percentile of RTTs (85%).

HTTP Keep-Alive	Overall Rate	Throughput	95th Bif	95th Rtt
enabled	4.78 <i>Gbps</i>	1.67 <i>Gbps</i>	0.21 <i>MB</i>	0.607 <i>ms</i>
disabled	4.56 <i>Gbps</i>	1.46 <i>Gbps</i>	0.16 <i>MB</i>	0.712 <i>ms</i>

Table B.1: A comparison of the average performance with and without HTTP Keep-Alive.

What Can Be Learned Using GMM Analysis To further analyze the impact of HTTP Keep-Alive on RPC performance, we then build two GMMs using the per-RPC metrics collected with Keep-Alive enabled or disabled, respectively. The 1d-Wasserstein distance is calculated between the blobs along different performance dimensions and the top five distinguishing performance metrics are presented in Table B.2. The results indicate that when HTTP Keep-Alive is disabled, the most distinguishable performance metrics are related to bytes-in-flight. This implies that the performance differences between the RPCs in different blobs are primarily due to volume, rather than delay and rate. However, when HTTP Keep-Alive is enabled, both RTT and bytes-in-

flight become the top distinguishable performance metrics. This suggests that the performance differences between the RPCs in different blobs are significant in both delay and volume.

	w/o HTTP Keep-Alive		with HTTP Keep-Alive	
	performance metric	distance	performance metric	distance
1	bif_p50	1.49	rtt_p75	1.40
2	bif_p75	1.41	rtt_p50	1.38
3	bif_p25	1.37	bif_p50	1.36
4	bif_p5	1.31	rtt_p95	1.34
5	bif_p95	1.27	bif_p75	1.32

Table B.2: Top 5 distinguishing performance metrics among blobs in each GMMs.

To make a fair comparison between the blobs in the two GMMs, we match them using the Wasserstein distance, as described in Section 4.2. Table B.3 displays the 2-dimensional Wasserstein distance between the blobs in these two GMMs. The blobs with the smallest distance are matched and labeled using the same legend ($P0$, $P1$, and $P2$) in the subsequent plots.

	$P0_{disable}$	$P1_{disable}$	$P2_{disable}$
$P0_{enable}$	5.27	12.65	12.79
$P1_{enable}$	8.34	2.04	16.13
$P2_{enable}$	35.36	7.97	7.76

Table B.3: Distance measure of the 2-dimensional Wasserstein distance between blobs in two GMMs.

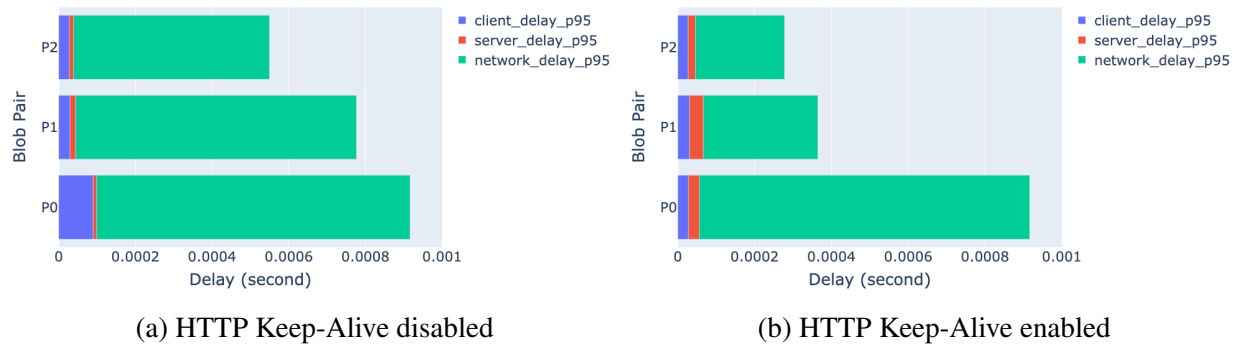


Figure B.1: Breakdown of delay on end-hosts and network for each blob with and without HTTP Keep-Alive in the GMMs with a single connection.

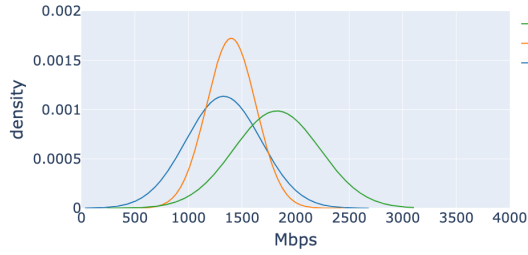
Figure B.1 shows the breakdown of the most notable delays on end hosts and in the network for each blob in the two GMMs. The majority of the delay occurs in the network for all the blobs,

but the network delay for the three blobs is similar when HTTP Keep-Alive is disabled (Figure B.1(a)) and varies significantly when it is enabled (Figure B.1(b)). This explains why delay metrics are only among the top five distinguishing performance metrics between blobs when HTTP Keep-Alive is enabled. When HTTP Keep-Alive is disabled, RPCs in blob $P0$, which represent 20% of the total RPCs, experience higher client delay compared to the others (Figure B.1(a)). Comparing the two GMMs reveals that when HTTP Keep-Alive is enabled, RPCs in $P1$ and $P2$ have lower network delay, indicating less severe constraints in the network. Meanwhile, RPCs in $P0$ have lower client delay, implying less constraints on the client side but no significant changes in network delay.

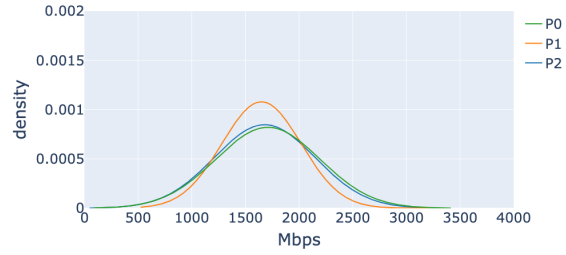
Figure B.2 presents the kde graphs comparing the most significant performance metrics between the matched blobs in the two GMMs using the Wasserstein distance. These metrics are: throughput, the 5th and 95th percentile of bytes-in-flight, and the 95th percentile of network delay. The comparison between the $P0$ blobs in the two GMMs shows that when HTTP Keep-Alive is enabled, the previously volume-constrained RPCs experience improved performance, as evidenced by their higher throughput. With HTTP Keep-Alive enabled, the bandwidth resources are distributed more equitably among the RPCs in the different blobs within one GMM. For instance, the throughput of RPCs in $P0$ decreases slightly from approximately 1825 Mbps to 1709 Mbps, while the throughput of RPCs in $P1$ and $P2$ increases from approximately 1390 Mbps to 1650 Mbps, leading to better overall performance.

In conclusion, the GMM analysis of the performance comparison shows that when there is at most one connection between each server/client pair in our testbed, enabling HTTP Keep-Alive leads to improved performance by optimizing bandwidth utilization, reducing delay, and increasing volume and rate. Furthermore, it results in a more uniform performance for all RPCs.

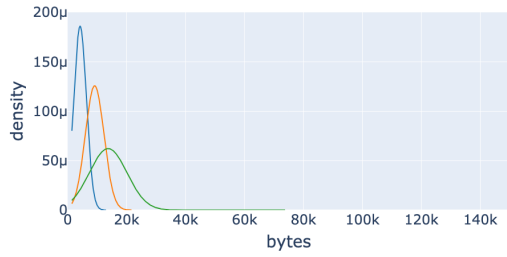
Validating Constraints with Ground-truth Information In this section, we validate the findings of our GMM analysis by using ground-truth information from sample pcap traces for each blob in the two GMMs with and without HTTP Keep-Alive enabled. Figure B.3 presents the time sequence plots of selected RPC traces from $P0$ and $P2$.



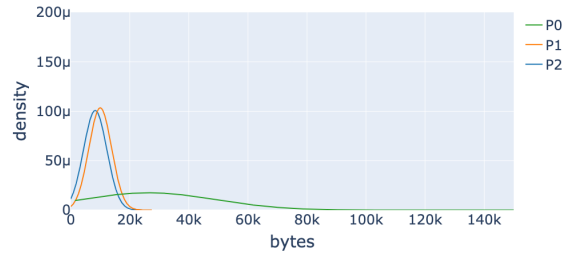
(a) *Disabled*: throughput (Mbps)



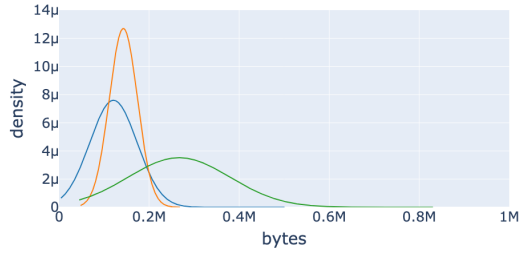
(b) *Enabled*: throughput (Mbps)



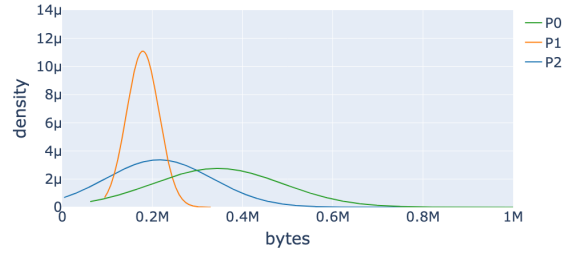
(c) *Disabled*: P5 bytes-in-flight



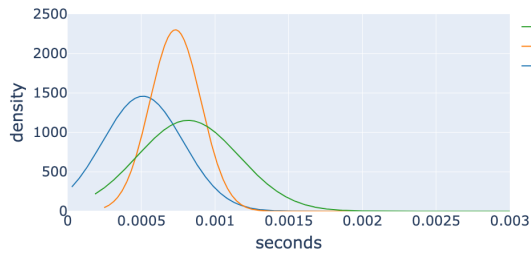
(d) *Enabled*: P5 bytes-in-flight



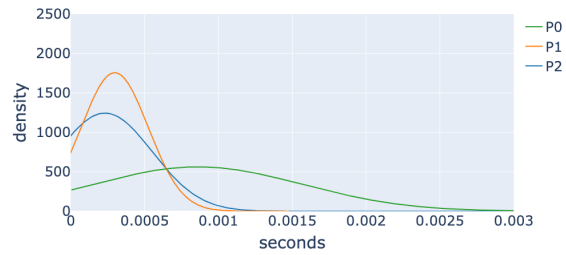
(e) *Disabled*: P95 bytes-in-flight



(f) *Enabled*: P95 bytes-in-flight



(g) *Disabled*: P95 network delay (sec)



(h) *Enabled*: P95 network delay (sec)

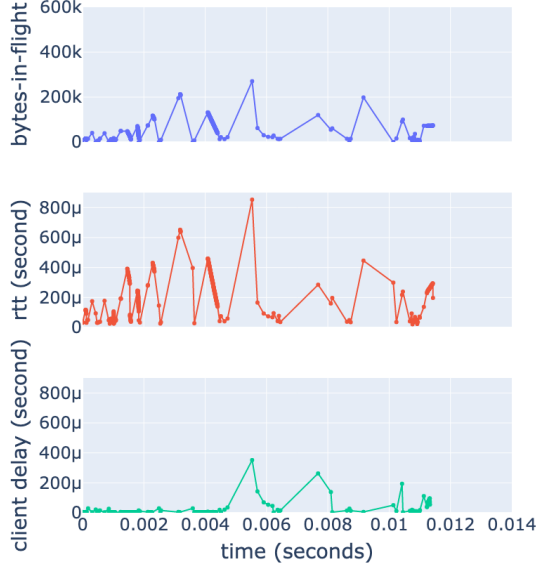
Figure B.2: Kernel density plots of throughput, bytes-in-flight, and network delay for multiple connections when HTTP Keep-Alive is disabled (*Disabled*: left column) and enabled (*Enabled*: right column).

When HTTP Keep-Alive is disabled, the RPC in $P0$ experiences a significant client delay during transmission, resulting in an increase in the measured RTT as observed in Figure B.3(a). However, with HTTP Keep-Alive enabled, the client delay becomes negligible and stable at around 20 microseconds, as shown in Figure B.3(b). This suggests that the main contributor to the increased RTTs in this scenario is the network, not the local hosts. Without HTTP Keep-Alive, the network performance of the RPCs is partially hindered by clients. With HTTP Keep-Alive, this constraint is relieved and the performance is mainly constrained by the network. It is also evident that without HTTP Keep-Alive, the increase in bytes-in-flight is slower due to the TCP slow start stage, as shown in Figure B.3(a), rather than starting at a high value at the beginning of the transfer, as observed in Figure B.3(b).

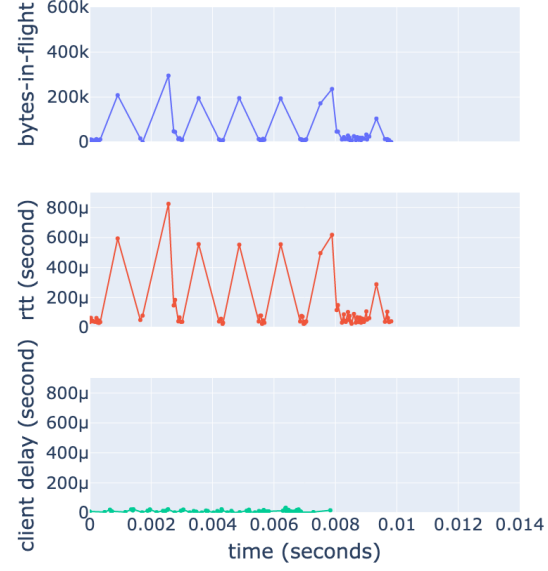
In the case of $P2$, the use of HTTP Keep-Alive leads to an increase in bytes-in-flight, resulting in a reduction of approximately 57% in the total transfer duration (as shown in Figure B.3(d)). When HTTP Keep-Alive is disabled, the RPC remains at a relatively low number of bytes-in-flight for most of the transmission (Figure B.3(c)), resulting in lower RTTs but a longer transfer duration and lower throughput. This indicates that the performance of the RPCs in $P2$ is mainly limited by volume and that enabling HTTP Keep-Alive greatly relieves this constraint. Further analysis of pcap traces reveals that there is more cross-traffic from multiple server/client pairs in the network when HTTP Keep-Alive is disabled. As shown in Figure B.4(b), more bytes are transmitted concurrently and it takes longer for the transfers to complete. This is consistent with the correlation between RTTs and bytes-in-flights, as shown in Figure B.4(a). When HTTP Keep-Alive is disabled, the same increase in bytes-in-flight leads to a larger increase in RTTs due to the greater number of bytes already in the network.

B.2 Experiment 2: The Impact of Communication Patterns

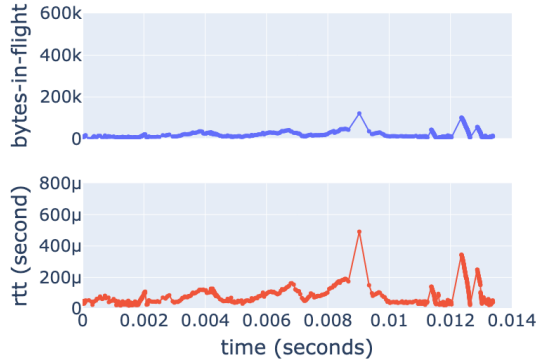
Finally, we examine the impact of different communication patterns on the performance of RPCs when there are multiple connections between each server/client with HTTP Keep-Alive enabled. Different communication patterns may cause different interactions among traffic and result



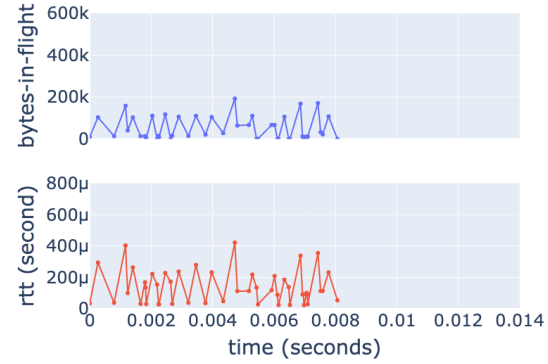
(a) $P0$: without HTTP Keep-Alive



(b) $P0$: with HTTP Keep-Alive

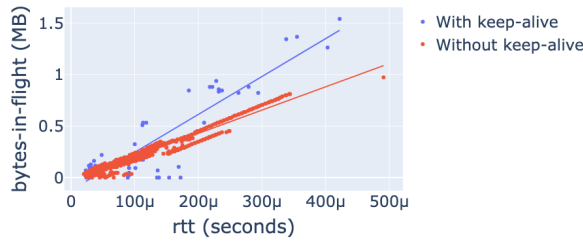


(c) $P2$: without HTTP Keep-Alive

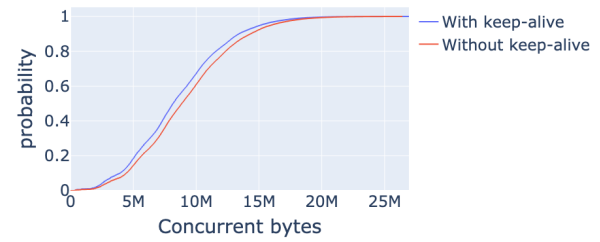


(d) $P2$: with HTTP Keep-Alive

Figure B.3: Time sequence plots of bytes-in-flight and round-trip times (RTTs) of selected traces from blobs $P0$ and $P2$ using multiple connections.



(a) Correlation between bytes-in-flight and RTT of RPCs in $P0$.



(b) Total bytes during transmission in the experiment based on pcap files.

Figure B.4: Ground truth information from pcap files.

in diverse queuing delays. To accurately assess the impact of these patterns on RPC network performance, we use GMMs. In contrast to the 1-to-1 communication scenario described in Section 5.4.2.1, where each client communicates with a specific server via multiple TCP connections, we examine a 1-to-5 scenario where each client communicates with five different servers, and correspondingly, each server communicates with five different clients through a single TCP connection. Each connection transferred 1 GB of data through multiple HTTP requests and responses. As a result, 70 concurrent TCP connections are established across the network, transferring a total of 70GB of data.

Figure B.5 shows the TCP-tuple information ($\langle src_ip : src_port - dst_ip : dst_port \rangle$) extracted from the pcap files on one server with these two different patterns. In the 1-to-1 pattern (Figure B.5(a)), the server (192.168.1.140) maintains five TCP connections to the same client (192.168.1.204), each with a different port number (39030, 39032, 39034, 39036, and 39038). In the 1-to-5 scenario, the server (192.168.1.13) communicates with five different clients (192.168.1.203, 192.168.1.206, 192.168.1.74, 192.168.1.77, and 192.168.1.79). In this experiment, each connection still transfers around 1 GB of data.

data_bytes		data_bytes	
tcp_tuple		tcp_tuple	
192.168.1.140:80-192.168.1.204:39030	1000754648	192.168.1.13:80-192.168.1.203:46380	1002699958
192.168.1.140:80-192.168.1.204:39032	999716544	192.168.1.13:80-192.168.1.206:45384	1003390223
192.168.1.140:80-192.168.1.204:39034	1001640992	192.168.1.13:80-192.168.1.74:35476	998915040
192.168.1.140:80-192.168.1.204:39036	1003281704	192.168.1.13:80-192.168.1.77:35412	1000411264
192.168.1.140:80-192.168.1.204:39038	998341208	192.168.1.13:80-192.168.1.79:60920	1001824165

(a) One server to a single client
(b) One server to multiple clients

Figure B.5: TCP tuple information from one server during experiments with different communication patterns.

We apply the same analysis method as in previous experiments, constructing two GMMs based on per-RPC performance metrics collected in each scenario. We then match the resulting blobs in the two GMMs using the 2d Wasserstein distance. The matched blobs are annotated

with the same legend: $P0$, $P1$, and $P2$. The distances between the matched blobs are 0.074 between $P0$, 0.036 between $P1$, and 0.031 between $P2$, which are much smaller compared to the minimum distance of 2.04 between the matched blobs in Table B.3 in Section B.1.

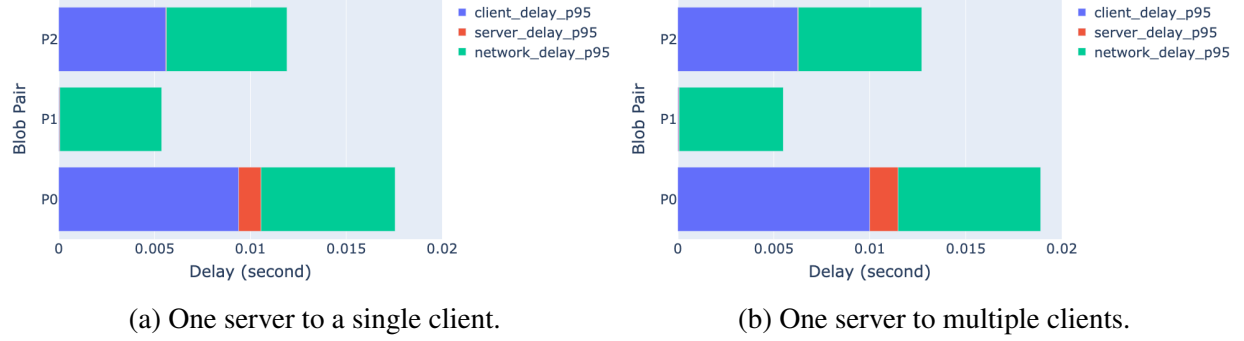


Figure B.6: Delay breakdown for blobs in each GMM with different communication patterns.

Figure B.6 shows the breakdown of the delay in each blob, displaying consistent performance on both local hosts and in the network. Since $P0$ has the largest distance among these three matched blobs, Table B.4 ranks their top 5 distinguishing performance metrics in two GMMs based on their distance. As can be seen, the maximum distance between the most distinctive performance metric (rtt_p95) is only around 0.106. Figure B.7 presents the kde plots of the most distinctive performance metrics in terms of delay, volume, and rate for $P0$, which consistently shows minimal differences.

	performance metric	distance
1	rtt_p95	0.106
2	rtt_p25	0.082
3	bif_p50	0.074
4	rtt_p95	0.074
5	throughput	0.052

Table B.4: Top 5 distinguishing performance metrics for $P0$.

Furthermore, categorical analysis did not indicate any substantial variations in the sizes of the RPCs and cross-traffic between matched blobs in the two GMMs. As depicted in Figure B.8, the percentage of RPC bytes in each blob is similar in the matched blobs, demonstrating that the volume of traffic is comparable.

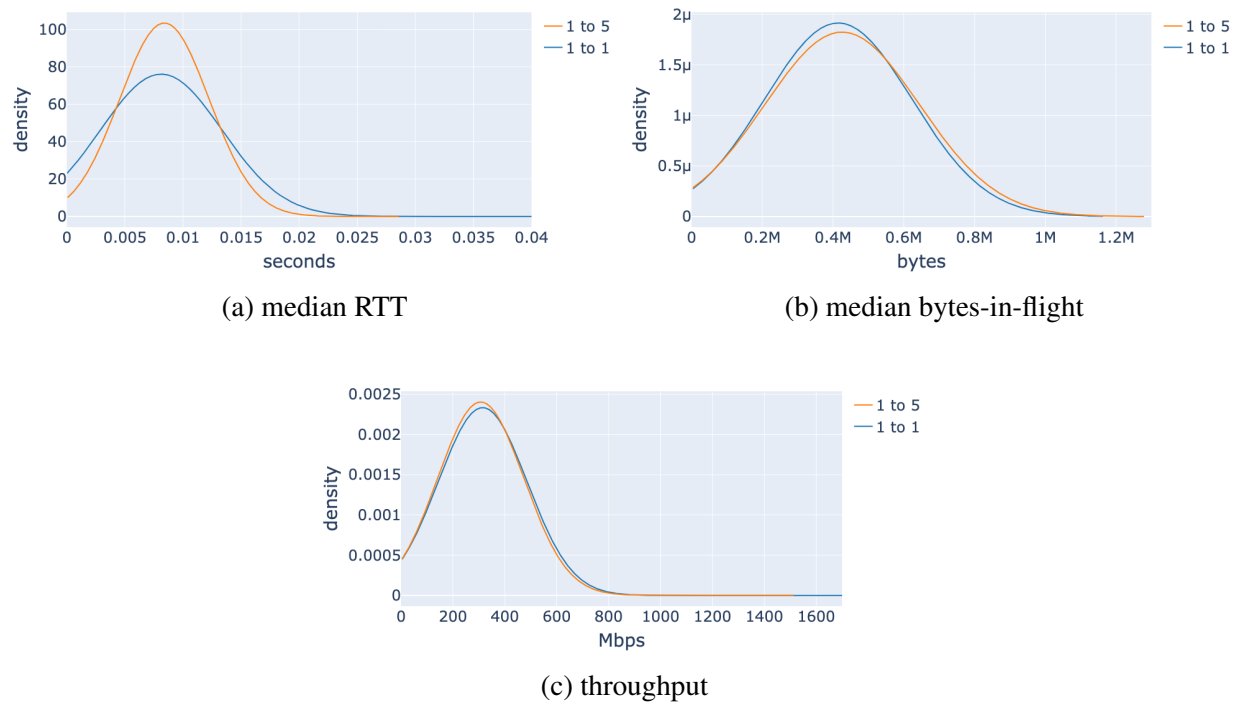


Figure B.7: KDE plots of $P0$ with different communication patterns: 1 to 1 vs. 1 to 5.

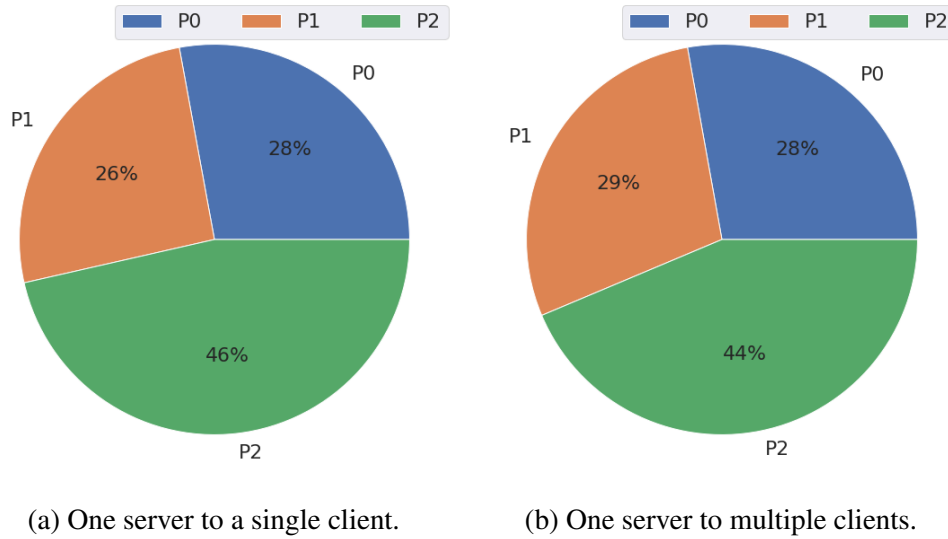


Figure B.8: The ratio of RPCs in each blob with different communication patterns.

In conclusion, our analysis using GMMs indicates that there are no notable variations between the two communication patterns. This finding suggests that, in our testbed with uniform hardware and software in clients and servers, and a balanced network topology, the specific choice of strategy for distributing RPC requests does not have a significant impact on the network performance of RPCs when the network workload is similar.

REFERENCES

- [1] “The hyperscale data center drives the global cloud revolution.” <https://dataconomy.com/2019/07/why-96-of-enterprises-face-ai-training-data-issues/>. Accessed: 2020-03-02.
- [2] A. A. Team, “Summary of the amazon ec2 and amazon rds service disruption in the us east region.” <https://aws.amazon.com/message/65648/>.
- [3] Wikipedia, “Microsoft azure.” https://en.wikipedia.org/wiki/Microsoft_Azure.
- [4] “Google cloud: Cloud computing services.” <https://cloud.google.com>, 2020 (accessed 2020-10-27).
- [5] Macrotrends, “Facebook net worth 2009-2021 | fb.” <https://www.macrotrends.net/stocks/charts/FB/facebook/net-worth>.
- [6] M. Iqbal, “Youtube revenue and usage statistics (2020).” <https://www.businessofapps.com/data/youtube-statistics/>.
- [7] “Stadia - one place for all the ways we play.” <https://stadia.google.com>, 2019.
- [8] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, K. N. B., C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev, S. Padgett, F. Rabe, S. Ray, M. Tewari, M. Tierney, M. Zahn, J. Zolla, J. Ong, and A. Vahdat, “B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google’s software-defined wan,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication, SIGCOMM ’18*, (New York, NY, USA), p. 74â87, Association for Computing Machinery, 2018.
- [9] Q. Zhang, G. Yu, C. Guo, Y. Dang, N. Swanson, X. Yang, R. Yao, M. Chintalapati, A. Krishnamurthy, and T. Anderson, “Deepview: Virtual disk failure diagnosis and pattern detection for azure,” in *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, pp. 519–532, 2018.
- [10] M. D. Dikaiakos, D. Katsaros, P. Mehra, G. Pallis, and A. Vakali, “Cloud computing: Distributed internet computing for it and scientific research,” *IEEE Internet computing*, vol. 13, no. 5, pp. 10–13, 2009.
- [11] L. Wang, J. Tao, R. Ranjan, H. Marten, A. Streit, J. Chen, and D. Chen, “G-hadoop: Mapreduce across distributed data centers for data-intensive computing,” *Future Generation Computer Systems*, vol. 29, no. 3, pp. 739–750, 2013.
- [12] Google, “grpc: A high performance, open source universal rpc framework.” <https://grpc.io/>, (accessed: 04.09.2021).

- [13] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol-http/1.1," tech. rep., 1999.
- [14] "Youtube." <https://www.youtube.com>, 2018-08-29 (accessed 2020-10-27).
- [15] "Netflix - watch tv shows online, watch movies online." <https://www.netflix.com>, 2020 (accessed 2020-10-27).
- [16] "Facebook." <https://www.facebook.com>, 2020 (accessed 2020-10-27).
- [17] "Gmail - google." <https://mail.google.com>, 2020 (accessed 2020-10-27).
- [18] "Maps - apple." <https://www.apple.com/maps/>, 2020 (accessed 2020-10-27).
- [19] C. Guo, L. Yuan, D. Xiang, Y. Dang, R. Huang, D. Maltz, Z. Liu, V. Wang, B. Pang, H. Chen, *et al.*, "Pingmesh: A large-scale system for data center network latency measurement and analysis," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 139–152, 2015.
- [20] Y. Zhu, N. Kang, J. Cao, A. Greenberg, G. Lu, R. Mahajan, D. Maltz, L. Yuan, M. Zhang, B. Y. Zhao, *et al.*, "Packet-level telemetry in large datacenter networks," in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 479–491, 2015.
- [21] A. Roy, H. Zeng, J. Bagga, and A. C. Snoeren, "Passive realtime datacenter fault detection and localization," in *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*, pp. 595–612, 2017.
- [22] A. Rao, A. Legout, Y.-s. Lim, D. Towsley, C. Barakat, and W. Dabbous, "Network characteristics of video streaming traffic," in *Proceedings of the seventh conference on emerging networking experiments and technologies*, pp. 1–12, 2011.
- [23] Z. Zhang and C. Williamson, "A campus-level view of outlook email traffic," in *Proceedings of the 2018 VII International Conference on Network, Communication and Computing*, pp. 299–306, 2018.
- [24] J. Johnson and J. Jeff, *GUI bloopers: don'ts and do's for software developers and Web designers*. Morgan Kaufmann, 2000.
- [25] F. F.-H. Nah, "A study on tolerable waiting time: how long are web users willing to wait?," *Behaviour & Information Technology*, vol. 23, no. 3, pp. 153–163, 2004.
- [26] J. D. Brutlag, H. Hutchinson, and M. Stone, "User preference and search engine latency," 2008.
- [27] R. Jota, A. Ng, P. Dietz, and D. Wigdor, "How fast is fast enough? a study of the effects of latency in direct-touch pointing tasks," in *Proceedings of the sigchi conference on human factors in computing systems*, pp. 2291–2300, 2013.

- [28] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, *et al.*, “Jupiter rising: A decade of clos topologies and centralized control in google’s datacenter network,” *ACM SIGCOMM computer communication review*, vol. 45, no. 4, pp. 183–197, 2015.
- [29] L. Poutievski, O. Mashayekhi, J. Ong, A. Singh, M. Tariq, R. Wang, J. Zhang, V. Beau-regard, P. Conner, S. Gribble, *et al.*, “Jupiter evolving: Transforming google’s datacenter network via optical circuit switches and software-defined networking,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, pp. 66–85, 2022.
- [30] O. Alipourfard, J. Gao, J. Koenig, C. Harshaw, A. Vahdat, and M. Yu, “Risk based planning of network changes in evolving data centers,” in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pp. 414–429, 2019.
- [31] Z. Li, Q. Cheng, K. Hsieh, Y. Dang, P. Huang, P. Singh, X. Yang, Q. Lin, Y. Wu, S. Levy, *et al.*, “Gandalf: An intelligent, end-to-end analytics service for safe deployment in large-scale cloud infrastructure,” in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, pp. 389–402, 2020.
- [32] N. Cardwell, S. Savage, and T. Anderson, “Modeling tcp latency,” in *Proceedings IEEE INFOCOM 2000. Conference on Computer Communications. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies (Cat. No. 00CH37064)*, vol. 3, pp. 1742–1751, IEEE, 2000.
- [33] M. Yu, A. G. Greenberg, D. A. Maltz, J. Rexford, L. Yuan, S. Kandula, and C. Kim, “Profiling network performance for multi-tier data center applications,” in *NSDI*, vol. 11, pp. 5–5, 2011.
- [34] R. N. Mysore, R. Mahajan, A. Vahdat, and G. Varghese, “Gestalt: Fast, unified fault localization for networked systems,” in *2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14)*, pp. 255–267, 2014.
- [35] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, “Seer: Leveraging big data to navigate the complexity of performance debugging in cloud microservices,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 19–33, 2019.
- [36] X. Wu, D. Turner, C.-C. Chen, D. A. Maltz, X. Yang, L. Yuan, and M. Zhang, “Netpilot: automating datacenter network failure mitigation,” in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pp. 419–430, 2012.
- [37] C. H. Wagner, “Simpson’s paradox in real life,” *The American Statistician*, vol. 36, no. 1, pp. 46–48, 1982.
- [38] M. A. Hernán, D. Clayton, and N. Keiding, “The simpson’s paradox unraveled,” *International journal of epidemiology*, vol. 40, no. 3, pp. 780–785, 2011.

- [39] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Japan, and C. Shanbhag, “Dapper, a large-scale distributed systems tracing infrastructure,” 2010.
- [40] B. Arzani, S. Ciraci, B. T. Loo, A. Schuster, and G. Outhred, “Taking the blame game out of data centers operations with netpoirot,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, pp. 440–453, 2016.
- [41] J. Li, W. Pedrycz, and I. Jamal, “Multivariate time series anomaly detection: A framework of hidden markov models,” *Applied Soft Computing*, vol. 60, pp. 229–240, 2017.
- [42] K. O’Shea and R. Nash, “An introduction to convolutional neural networks,” *arXiv preprint arXiv:1511.08458*, 2015.
- [43] M. Schuster and K. K. Paliwal, “Bidirectional recurrent neural networks,” *IEEE transactions on Signal Processing*, vol. 45, no. 11, pp. 2673–2681, 1997.
- [44] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [45] H. Ren, B. Xu, Y. Wang, C. Yi, C. Huang, X. Kou, T. Xing, M. Yang, J. Tong, and Q. Zhang, “Time-series anomaly detection service at microsoft,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 3009–3017, 2019.
- [46] Y. Su, Y. Zhao, C. Niu, R. Liu, W. Sun, and D. Pei, “Robust anomaly detection for multivariate time series through stochastic recurrent neural network,” in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 2828–2837, 2019.
- [47] C. Zhang, D. Song, Y. Chen, X. Feng, C. Lumezanu, W. Cheng, J. Ni, B. Zong, H. Chen, and N. V. Chawla, “A deep neural network for unsupervised anomaly detection and diagnosis in multivariate time series data,” in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 1409–1416, 2019.
- [48] D. A. Reynolds, “Gaussian mixture models.,” *Encyclopedia of biometrics*, vol. 741, 2009.
- [49] S.-w. Liao, T.-H. Hung, D. Nguyen, C. Chou, C. Tu, and H. Zhou, “Machine learning-based prefetch optimization for data center applications,” in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pp. 1–10, 2009.
- [50] M. Zaharia, R. S. Xin, P. Wendell, T. Das, M. Armbrust, A. Dave, X. Meng, J. Rosen, S. Venkataraman, M. J. Franklin, *et al.*, “Apache spark: a unified engine for big data processing,” *Communications of the ACM*, vol. 59, no. 11, pp. 56–65, 2016.
- [51] R. Ricci, E. Eide, and C. Team, “Introducing cloudlab: Scientific infrastructure for advancing cloud architectures and applications,” ; *login:: the magazine of USENIX & SAGE*, vol. 39, no. 6, pp. 36–38, 2014.

- [52] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren, “Inside the social network’s (datacenter) network,” in *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pp. 123–137, 2015.
- [53] J. D. Case, M. Fedor, M. L. Schoffstall, and J. Davin, “Rfc1157: Simple network management protocol (snmp),” 1990.
- [54] M. Kerrisk, “Ifconfig(8) – linux system administrator’s manual.” <https://man7.org/linux/man-pages/man8/ifconfig.8.html>.
- [55] V. Jacobson, “Tcpdump,” *ftp://ftp. ee. lbl. gov*, 1989.
- [56] M. A. Qureshi, Y. Cheng, Q. Yin, Q. Fu, G. Kumar, M. Moshref, J. Yan, V. Jacobson, D. Wetherall, and A. Kabbani, “Plb: congestion signals are simple and effective for network load balancing,” in *Proceedings of the ACM SIGCOMM 2022 Conference*, pp. 207–218, 2022.
- [57] A. Aizawa, “An information-theoretic perspective of tf–idf measures,” *Information Processing & Management*, vol. 39, no. 1, pp. 45–65, 2003.
- [58] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009.
- [59] R. Fonseca, G. Porter, R. H. Katz, and S. Shenker, “X-trace: A pervasive network tracing framework,” in *4th {USENIX} Symposium on Networked Systems Design & Implementation ({NSDI} 07)*, 2007.
- [60] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, “Google-wide profiling: A continuous profiling infrastructure for data centers,” *IEEE micro*, vol. 30, no. 4, pp. 65–79, 2010.
- [61] “Jaeger: open source, end-to-end distributed tracing.” <https://www.jaegertracing.io/>, (accessed 2020-10-27).
- [62] “Openzipkin: A distributed tracing system.” <https://zipkin.io/>, (accessed 2020-10-27).
- [63] J. Kaldor, J. Mace, M. Bejda, E. Gao, W. Kuropatwa, J. O’Neill, K. W. Ong, B. Schaller, P. Shan, B. Viscomi, *et al.*, “Canopy: An end-to-end performance tracing and analysis system,” in *Proceedings of the 26th Symposium on Operating Systems Principles*, pp. 34–50, 2017.
- [64] P. Las-Casas, G. Papakerashvili, V. Anand, and J. Mace, “Sifter: Scalable sampling for distributed traces, without feature engineering,” in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 312–324, 2019.
- [65] S. Floyd, “Tcp and explicit congestion notification,” *ACM SIGCOMM Computer Communication Review*, vol. 24, no. 5, pp. 8–23, 1994.

- [66] N. Jarvey, “Cbs all access crashes as super bowl lv kicks off.” <https://www.hollywoodreporter.com/business/digital/cbs-all-access-crashes-as-super-bowl-lv-kicks-off-4129254/>, 2021 (accessed: 04.09.2021).
- [67] J. Peters, “Prolonged aws outage takes down a big chunk of the internet.” <https://www.theverge.com/2020/11/25/21719396/amazon-web-services-aws-outage-down-internet>.
- [68] S. Liu, M. Yamada, N. Collier, and M. Sugiyama, “Change-point detection in time-series data by relative density-ratio estimation,” *Neural Networks*, vol. 43, pp. 72–83, 2013.
- [69] N. Laptev, S. Amizadeh, and I. Flint, “Generic and scalable framework for automated time-series anomaly detection,” in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 1939–1947, 2015.
- [70] A. A. Qahtan, B. Alharbi, S. Wang, and X. Zhang, “A pca-based change detection framework for multidimensional data streams: Change detection in multidimensional data streams,” in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pp. 935–944, 2015.
- [71] S. Aminikhanghahi and D. J. Cook, “A survey of methods for time series change point detection,” *Knowledge and information systems*, vol. 51, no. 2, pp. 339–367, 2017.
- [72] J. Mazel, P. Casas, R. Fontugne, K. Fukuda, and P. Owezarski, “Hunting attacks in the dark: clustering and correlation analysis for unsupervised anomaly detection,” *International Journal of Network Management*, vol. 25, no. 5, pp. 283–305, 2015.
- [73] L. Wei and E. Keogh, “Semi-supervised time series classification,” in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 748–753, 2006.
- [74] S. Reddy, M. Mun, J. Burke, D. Estrin, M. Hansen, and M. Srivastava, “Using mobile phones to determine transportation modes,” *ACM Transactions on Sensor Networks (TOSN)*, vol. 6, no. 2, pp. 1–27, 2010.
- [75] Y. Zheng, Y. Chen, Q. Li, X. Xie, and W.-Y. Ma, “Understanding transportation modes based on gps data for web applications,” *ACM Transactions on the Web (TWEB)*, vol. 4, no. 1, pp. 1–36, 2010.
- [76] M. Han, Y.-K. Lee, S. Lee, *et al.*, “Comprehensive context recognizer based on multimodal sensors in a smartphone,” *Sensors*, vol. 12, no. 9, pp. 12588–12605, 2012.
- [77] K. D. Feuz, D. J. Cook, C. Rosasco, K. Robertson, and M. Schmitter-Edgecombe, “Automated detection of activity transitions for prompting,” *IEEE transactions on human-machine systems*, vol. 45, no. 5, pp. 575–585, 2014.

- [78] Y. Kawahara and M. Sugiyama, “Sequential change-point detection based on direct density-ratio estimation,” *Statistical Analysis and Data Mining: The ASA Data Science Journal*, vol. 5, no. 2, pp. 114–127, 2012.
- [79] M. Yamada, A. Kimura, F. Naya, and H. Sawada, “Change-point detection with feature selection in high-dimensional time-series data,” in *Twenty-Third International Joint Conference on Artificial Intelligence*, 2013.
- [80] J. Benesty, J. Chen, Y. Huang, and I. Cohen, “Pearson correlation coefficient,” in *Noise reduction in speech processing*, pp. 1–4, Springer, 2009.
- [81] J. Dean and S. Ghemawat, “Mapreduce: simplified data processing on large clusters,” *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.
- [82] J. C. De Winter, “Using the student’s t-test with extremely small sample sizes,” *Practical Assessment, Research, and Evaluation*, vol. 18, no. 1, p. 10, 2013.
- [83] F. J. Massey Jr, “The kolmogorov-smirnov test for goodness of fit,” *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.
- [84] M. L. McHugh, “The chi-square test of independence,” *Biochemia medica*, vol. 23, no. 2, pp. 143–149, 2013.
- [85] B. D. Ripley, *Pattern recognition and neural networks*. Cambridge university press, 2007.
- [86] C. Delimitrou and C. Kozyrakis, “ibench: Quantifying interference for datacenter applications,” in *2013 IEEE international symposium on workload characterization (IISWC)*, pp. 23–33, IEEE, 2013.
- [87] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.
- [88] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, “Evolve or die: High-availability design principles drawn from googles network infrastructure,” in *Proceedings of the 2016 ACM SIGCOMM Conference, SIGCOMM ’16*, (New York, NY, USA), p. 58â72, Association for Computing Machinery, 2016.
- [89] C. Chatfield, “The holt-winters forecasting procedure,” *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, vol. 27, no. 3, pp. 264–279, 1978.
- [90] H. Abdi, “Discriminant correspondence analysis,” 2007.
- [91] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. A. Maltz, and M. Zhang, “Towards highly reliable enterprise network services via inference of multi-level dependencies,” *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 4, pp. 13–24, 2007.
- [92] M. J. Wainwright, M. I. Jordan, *et al.*, “Graphical models, exponential families, and variational inference,” *Foundations and Trends® in Machine Learning*, vol. 1, no. 1–2, pp. 1–305, 2008.

- [93] S. Kandula, R. Mahajan, P. Verkaik, S. Agarwal, J. Padhye, and P. Bahl, “Detailed diagnosis in enterprise networks,” in *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pp. 243–254, 2009.
- [94] R. R. Kompella, J. Yates, A. Greenberg, and A. C. Snoeren, “Fault localization via risk modeling,” *IEEE Transactions on Dependable and Secure Computing*, vol. 7, no. 4, pp. 396–409, 2009.
- [95] R. Tibshirani, “Regression shrinkage and selection via the lasso,” *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 58, no. 1, pp. 267–288, 1996.
- [96] G. Casella and R. L. Berger, *Statistical inference*. Cengage Learning, 2021.
- [97] J. Postel, “Rfc0793: Transmission control protocol,” 1981.
- [98] J. Postel *et al.*, “User datagram protocol,” 1980.
- [99] N. Cardwell, Y. Cheng, C. S. Gunn, S. H. Yeganeh, and V. Jacobson, “Bbr: Congestion-based congestion control,” *Queue*, vol. 14, no. 5, pp. 20–53, 2016.
- [100] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, “An architecture for differentiated services,” tech. rep., 1998.
- [101] A. Saeed, N. Dukkupati, V. Valancius, V. The Lam, C. Contavalli, and A. Vahdat, “Carousel: Scalable traffic shaping at end hosts,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pp. 404–417, 2017.
- [102] P. Sreekumari and J.-i. Jung, “Transport protocols for data center networks: a survey of issues, solutions and challenges,” *Photonic Network Communications*, vol. 31, no. 1, pp. 112–128, 2016.
- [103] 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.
- [104] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *Proceedings of the ACM SIGCOMM 2010 Conference*, pp. 63–74, 2010.
- [105] T. K. Moon, “The expectation-maximization algorithm,” *IEEE Signal processing magazine*, vol. 13, no. 6, pp. 47–60, 1996.
- [106] C. Rudin, “Stop explaining black box machine learning models for high stakes decisions and use interpretable models instead,” *Nature Machine Intelligence*, vol. 1, no. 5, pp. 206–215, 2019.
- [107] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, “Large-scale cluster management at google with borg,” in *Proceedings of the Tenth European Conference on Computer Systems*, pp. 1–17, 2015.
- [108] M. Rouaud, “Probability, statistics and estimation,” *Propagation of uncertainties*, 2013.

- [109] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, I. Stoica, *et al.*, “Spark: Cluster computing with working sets,” *HotCloud*, vol. 10, no. 10-10, p. 95, 2010.
- [110] J. Matoušek and B. Gärtner, *Understanding and Using Linear Programming*. Springer Science & Business Media, 2007.
- [111] R. Bellman, “Dynamic programming,” *Science*, vol. 153, no. 3731, pp. 34–37, 1966.
- [112] M. Ringnér, “What is principal component analysis?,” *Nature biotechnology*, vol. 26, no. 3, pp. 303–304, 2008.
- [113] L. v. d. Maaten and G. Hinton, “Visualizing data using t-sne,” *Journal of machine learning research*, vol. 9, no. Nov, pp. 2579–2605, 2008.
- [114] L. McInnes, J. Healy, and J. Melville, “Umap: Uniform manifold approximation and projection for dimension reduction,” *arXiv preprint arXiv:1802.03426*, 2018.
- [115] J. Li, K. Cheng, S. Wang, F. Morstatter, R. P. Trevino, J. Tang, and H. Liu, “Feature selection: A data perspective,” *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–45, 2017.
- [116] C. M. Jarque and A. K. Bera, “A test for normality of observations and regression residuals,” *International Statistical Review/Revue Internationale de Statistique*, pp. 163–172, 1987.
- [117] S. Sanders and J. Kaur, “Can web pages be classified using anonymized tcp/ip headers?,” in *Proceedings of IEEE INFOCOM*, IEEE, 2015.
- [118] S. I. Vrieze, “Model selection and psychological theory: a discussion of the differences between the akaike information criterion (aic) and the bayesian information criterion (bic).,” *Psychological methods*, vol. 17, no. 2, p. 228, 2012.
- [119] E.-J. Wagenmakers and S. Farrell, “Aic model selection using akaike weights,” *Psychonomic bulletin & review*, vol. 11, no. 1, pp. 192–196, 2004.
- [120] L. Lovmar, A. Ahlford, M. Jonsson, and A.-C. Syvänen, “Silhouette scores for assessment of snp genotype clusters,” *BMC genomics*, vol. 6, no. 1, p. 35, 2005.
- [121] I. J. Myung, “Tutorial on maximum likelihood estimation,” *Journal of mathematical Psychology*, vol. 47, no. 1, pp. 90–100, 2003.
- [122] C. M. Bishop and N. M. Nasrabadi, *Pattern recognition and machine learning*, vol. 4. Springer, 2006.
- [123] J. Sklansky, “Finding the convex hull of a simple polygon,” *Pattern Recognition Letters*, vol. 1, no. 2, pp. 79–83, 1982.
- [124] M. Thomas and A. T. Joy, *Elements of information theory*. Wiley-Interscience, 2006.

- [125] R. De Maesschalck, D. Jouan-Rimbaud, and D. L. Massart, “The mahalanobis distance,” *Chemometrics and intelligent laboratory systems*, vol. 50, no. 1, pp. 1–18, 2000.
- [126] “Leverage (statistics).” [https://en.wikipedia.org/wiki/Leverage_\(statistics\)#Determination_of_outliers_in_X_using_leverages](https://en.wikipedia.org/wiki/Leverage_(statistics)#Determination_of_outliers_in_X_using_leverages).
- [127] D. J. Olive, “Applied robust statistics,” *Preprint M-02-006*, 2008.
- [128] Y. B. Nikolay Laptev, Saeed Amizadeh, “A benchmark dataset for time series anomaly detection.” <https://yahoooresearch.tumblr.com/post/114590420346/a-benchmark-dataset-for-time-series-anomaly>.
- [129] N. Fournier and A. Guillin, “On the rate of convergence in wasserstein distance of the empirical measure,” *Probability Theory and Related Fields*, vol. 162, no. 3-4, pp. 707–738, 2015.
- [130] A. Ramdas, N. García Trillos, and M. Cuturi, “On wasserstein two-sample testing and related families of nonparametric tests,” *Entropy*, vol. 19, no. 2, p. 47, 2017.
- [131] R. M. Gray, *Entropy and information theory*. Springer Science & Business Media, 2011.
- [132] H. V. Nguyen and L. Bai, “Cosine similarity metric learning for face verification,” in *Asian conference on computer vision*, pp. 709–720, Springer, 2010.
- [133] J. Delon and A. Desolneux, “A wasserstein-type distance in the space of gaussian mixture models,” *SIAM Journal on Imaging Sciences*, vol. 13, no. 2, pp. 936–970, 2020.
- [134] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, *et al.*, “{TAO}: Facebook’s distributed data store for the social graph,” in *2013 {USENIX} Annual Technical Conference ({USENIX}{ATC} 13)*, pp. 49–60, 2013.
- [135] C. Clos, “A study of non-blocking switching networks,” *Bell System Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.
- [136] R. Soni, *Nginx*. Springer, 2016.
- [137] G. W. Mike Hibler, Leigh Stoller, “Time synchronization at cloudlab clusters.” <https://gitlab.flux.utah.edu/emulab/emulab-devel/-/issues/658>.
- [138] P. Sirinam, M. Imani, M. Juarez, and M. Wright, “Deep fingerprinting: Undermining website fingerprinting defenses with deep learning,” in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1928–1943, ACM, 2018.
- [139] K. Krishna and M. N. Murty, “Genetic k-means algorithm,” *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 29, no. 3, pp. 433–439, 1999.
- [140] E. Schubert, J. Sander, M. Ester, H. P. Kriegel, and X. Xu, “Dbscan revisited, revisited: why and how you should (still) use dbscan,” *ACM Transactions on Database Systems (TODS)*, vol. 42, no. 3, pp. 1–21, 2017.

- [141] A. D. Birrell and B. J. Nelson, “Implementing remote procedure calls,” *ACM Transactions on Computer Systems (TOCS)*, vol. 2, no. 1, pp. 39–59, 1984.
- [142] K. Ramakrishnan, S. Floyd, and D. Black, “The addition of explicit congestion notification (ecn) to ip,” tech. rep., 2001.
- [143] W. de Bruijn, “net-timestamp: new tx timestamps and tcp (linux 3.17).” <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=618896e6d00773d6d50e0b19f660af22fa26cd61>.
- [144] T. Dunning and O. Ertl, “Computing extremely accurate quantiles using t-digests,” *arXiv preprint arXiv:1902.04023*, 2019.
- [145] T. Benson, A. Anand, A. Akella, and M. Zhang, “Understanding data center traffic characteristics,” *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 1, pp. 92–99, 2010.
- [146] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, 2010.
- [147] Q. Yin, J. Kaur, and F. D. Smith, “Tcp rapid: from theory to practice,” in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pp. 1–9, IEEE, 2017.
- [148] A. Kabbani, B. Vamanan, J. Hasan, and F. Duchene, “Flowbender: Flow-level adaptive routing for improved latency and throughput in datacenter networks,” in *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pp. 149–160, 2014.