TcpRT: Instrument and Diagnostic Analysis System for Service Quality of Cloud Databases at Massive Scale in Real-time

Wei Cao, Yusong Gao, Bingchen Lin, Xiaojie Feng, Yu Xie, Xiao Lou, Peng Wang

Alibaba Cloud

{mingsong.cw,jianchuan.gys,bingchen.lbc,xiaojie.fxj,qianqing.xy,louxiao.lx,wangpeng.wangp}@alibaba-inc.com

ABSTRACT

Smooth end-to-end performance of mission-critical database system is essential to the stability of applications deployed on the cloud. It's a challenge for cloud database vendors to detect any performance degradation in real-time and locate the root cause quickly in sophisticated network environment. Cloud databases vendors tend to favor a multi-tier distributed architecture to achieve multi-tenant management, scalability and high-availability, which may further complicate the problem.

This paper presents TcpRT, the instrument and diagnosis infrastructure in Alibaba Cloud RDS that achieves real-time anomaly detection. We wrote a Linux kernel module to collect trace data of each SQL query, designed to be efficient with minimal overhead, it adds tracepoints in callbacks of TCP congestion control kernel module, that is totally transparent to database processes. In order to reduce the amount of data significantly before sending it to backend, raw trace data is aggregated. Aggregated trace data is then processed, grouped and analyzed in a distributed streaming computing platform. By utilizing a self-adjustable Cauchy distribution statistical model from historical performance data for each DB instance, anomalous events can be automatically detected in databases, which eliminates manually configuring thresholds by experience. A fault or hiccup occurred in any network component that is shared among multiple DB instances (e.g. hosted on the same physical machine or uplinked to the same pair of TOR switches) may cause large-scale service quality degradations. The ratio of anomalous DB instances vs networks components is being calculated, which helps pinpoint the faulty component.

TcpRT has been deployed in production at Alibaba Cloud for the past 3 years, collects over 20 million raw traces per second, and processes over 10 billion locally aggregated results in the backend per day, and managed to have within 1% performance impact on DB system. We present case studies of typical scenarios where TcpRT helps to solve various problems occurred in production system.

CCS CONCEPTS

- **Information systems** → **Autonomous database administration**;
- **Networks** → **Network monitoring**; *Transport protocols*;
- **Computing methodologies** → **Anomaly detection**;

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

SIGMOD'18, June 10-15, 2018, Houston, TX, USA

© 2018 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-4703-7/18/06.

<https://doi.org/10.1145/3183713.3190659>

KEYWORDS

Cloud database; Database-as-a-Service (DBaaS); Service level agreement (SLA); Diagnosis; JStorm; Redis; Kafka; HybridDB; Alibaba Cloud

ACM Reference Format:

Wei Cao, Yusong Gao, Bingchen Lin, Xiaojie Feng, Yu Xie, Xiao Lou, Peng Wang. 2018. TcpRT: Instrument and Diagnostic Analysis System for Service Quality of Cloud Databases at Massive Scale in Real-time. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10-15, 2018, Houston, TX, USA*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3183713.3190659>

1 INTRODUCTION

Cloud database is a fast growing cloud service provided by many cloud computing vendors, like AWS RDS, Azure Cloud SQL Database, Google Cloud SQL, and Alibaba Cloud RDS. As more and more enterprises are moving applications and data from on-premises infrastructure environments to the cloud, cloud database has become the key component of the system, and the reliability of cloud database is vital for the stability of customers' businesses. Decline in service quality of mission critical database could cause user application to suffer from severe performance degradation, so it's essential for cloud database vendors to make commitments to users with a SLA, and provide instrumentation data to tell whether service level objects (SLOs) are met.

Commercial cloud databases of major vendors are powered by in-house multi-tenant DBaaS platforms, which typically have a distributed architecture [5, 10]. User applications are deployed in virtual machines and access the endpoint of database service through virtualized network, there are multiple layers of physical or software-defined network nodes between them, for example Top-Of-Rack (TOR) switches, routers, virtual switches, L4 load balancer, and SQL gateways (a.k.a. proxy) [5]. Many factors may influence database performance in such a complicated network topology, such as packet loss, TCP incast [18], OS kernel faults and slow disks, so troubleshooting can be quite difficult. The ability of quickly diagnosing and solving database service quality degradations is critical for cloud database vendors, which helps decrease Mean-Time-To-Recover (MTTR) and improve Recovery Time Objective (RTO).

Query latency is one of the most important database performance metrics. Unlike traditional (on-premises) database system, users care more about the end-to-end latency of DBaaS service including the cost on network and middle tiers, than just query processing time of the RDBMS backend itself. It's important to add traces to distinguish the time spent on DB node, proxy node and

the network separately. Network overhead should be further decomposed into different links, specifically, user-to-proxy, proxy-to-DB, DB-to-proxy, and proxy-to-user. Network anomalies, for example, TCP retransmissions, jitters in RTT, and RST packets should also be captured, since they are strong evidences of network issues and useful for diagnosis.

Prior distributed tracing systems, like Dapper [16] and Canopy [9] are intrusive to the application, each of them provides a run-time library which is intended to be incorporated into applications. These stub libraries generates and emits events when trace points are triggered during processing requests. To avoid any significant impact on performance, only a small fraction of requests are traced and recorded, adaptive sampling algorithms are proposed in both Dapper and Canopy. This solution may be suitable for in-house systems, however, inappropriate in our situations. As it's not feasible to modify users' application code base. Furthermore, sampling tracing method may also miss hiccup or faults that occurred in a very short period due to likelihood.

This paper describes the design and implementation of TcpRT, the instrument and diagnostic analysis system for service quality of Alibaba Cloud RDS. TcpRT proposes a novel method to collect tracing data related to DB instances in a non-intrusive and low-cost way, the collector is deployed to DBaaS servers like proxy and DB node, which gathers trace events from a set of selected callbacks in TCP congestion control kernel module. A lot of useful metrics can then be calculated from these events, including the end-to-end latency of a SQL request and time consumed in each stage. Other than various latency metrics, network quality of each TCP connection to database can also be quantified, by counting the number of anomaly events the collector, e.g., incoming packets with out-of-order TCP sequence number, TCP retransmission, jitters in RTT, RST packets and so on, this information is quite hard to be acquired outside the OS kernel. TcpRT collector works inside the kernel space and is totally transparent to DBaaS processes running in user space, without any code modification to them. It also has little impact on database performance, and both the increment of latency and the drop of throughput are within 1% when TcpRT collector is enabled.

Sudden increases/drops of key metrics are strong signals of service quality reduction, e.g., if a large part of SQL queries are getting much slower to response and the number of active sessions increases significantly within a short time, the probability of database service becoming unhealthy rises. Usually cloud vendors allow expert database administrators hired by enterprises to set some thresholds on important metrics manually, and alarms will be triggered when the value of metrics jump out of the expected range, i.e., the threshold of query response time can be set, so that any SQL query executed longer than this value is recognized as slow query. However, these thresholds are specific to applications and may be changed as time goes by, e.g., some databases are intended to perform analytic queries, it's not surprising to have big queries longer than dozens of seconds or even tens of minutes, there would be too many false alarm if we configure it with a threshold value customized for OLTP databases. Further more, there are hundreds of thousands of DB instances on our DBaaS platform, it's impossible to configure threshold manually for each of them.

TcpRT uses self-adjustable threshold settings trained from historical performance data of each DB instance. A Cauchy distribution statistical model is built from the historical time series, which is used to determine whether incoming metric value is within the expected range. Cauchy distribution is proven to be much more stable than normal distribution in existence of some significant outliers.

Gateway servers (e.g. proxy, load balancer) and physical resources (e.g. network, machine) could be shared among multiple instances in Database-as-a-Service (DBaaS) platform. Because each instance is independent of each other, it should be very rare to see the key metrics of them increase/drop suddenly at the same pace. If this phenomenon happens, it's most likely that they are influenced by the variation of some common resource they share. For example, if all instances on the same machine are observed to have sharp drop of throughput or significantly increase in latency at the same time, it's a strong evidence that there is something wrong with the underlying machine, perhaps faulty hardware or kernel bug etc. Similarly, if a large portion of instances uplinked to the same network component (e.g. TOR, proxy, load balancer) are abnormal, we can deduce the network component is suspicious. In other words, conformity behavior of a crowd of anomalous instances associated physically is a strong signal to a malfunctional common component, which is a simple yet pragmatic rule for root cause analysis.

TcpRT collects enormous amount of data and analyze it in real-time, the amount of data is still increasing with a growing number of instances, it's a challenge to manage the scalability and fault resilience of such a distributed stream computing system. JStorm[3] is a streaming process engine widely used inside Alibaba, and we developed TcpRT's analytical jobs on JStorm. A lot of optimizations are made to reduce the computing overhead. A new shuffle strategy, *local field grouping*, is added to decrease the amount of data shuffled and transferred between nodes by an order of magnitude. Backpressure is tuned more gracefully to overcome spikes in traffic. Finally, analysis results are written to a short-term KV-cache and persisted to a long-term analytic database later.

TcpRT has been deployed in production at Alibaba Cloud for the past 3 years, collects over 20 million raw traces per second, and processes over 10 billion locally aggregated results in the backend everyday. The contributions of this paper are following:

- We describe a novel technique to collect traces of each SQL query, which is implemented on top of the network congestion module in Linux kernel, and tracks the latency of each request to RDBMS and the end-to-end service reliability of each network connection in a non-intrusive and low-cost way.
- We built a complete stream computing system to analyze the data, optimized for scalability and fault resilience, processing events in real-time, and supporting exact-once semantics.
- We describe a novel analysis algorithm which utilizes TcpRT data to detect performance anomalies of RDBMS, and locate the root cause. We provide some typical scenarios where TcpRT helps diagnose problems occurred in production system.

2 BACKGROUND AND MOTIVATION

2.1 The DBaaS Platform

Alibaba Cloud RDS is a multi-tenant DBaaS platform which provides automatic provisioning of database instances and the ability to scale them on-demand. We use state-of-art resource isolation techniques to deploy and manage instances on physical machines, i.e., light-weight KVM with tailored image, and docker container. Generally speaking, virtual machine takes up more system resources but has better isolation than container.

As shown in Figure 1, Alibaba Cloud RDS network platform is composed of two major components, the control plane and the data plane. The control plane contains a collection of managers, including resource manager, high-availability (HA) manager, migration manager and so on. There are at least two replicas across multi-availability-zones (multi-AZs) for each DB instance, the resource manager is in charge of assigning replicas of instances to distinct physical machines. HA manager will detect failure of the leader replica and switch the service endpoint to the standby replica, in order to minimize the service unavailable time. Migration manager would migrate database processes from one machine to another, if loads are unbalanced among machines.

Figure 1: The Cloud database service network architecture.

There are multiple tiers on the data plane. User applications are deployed in Elastic Compute Service (ECS) and interconnect with RDS instances via Virtual Private Cloud (VPC). vSwitches reside in customer's VPC network, packets are packed/unpacked and routed by vSwitches. Service Load Balancer (SLB) can interpret packets coming from vSwitches, and map DBaaS IP endpoints to real servers. Incoming traffic is distributed evenly among multiple real servers. SLB supports both FNAT mode and DNAT mode. With FNAT mode, both incoming and outgoing traffic travel through SLB. While in DNAT mode, SLB monitors the incoming traffic and outgoing traffic that is sent back directly from real servers to vSwitches. DNAT has better performance over FNAT due to outgoing traffic packet is sent back directly to origin. According to our observation, outgoing traffic is about 3-4 times larger than incoming traffic since queries form a large part of the requests to database. Next-Generation Load Balancer (NGLB) is introduced and aimed for optimization at the next level. Only the first packet (a SYN packet) of each TCP connection is processed by NGLB, vSwitches can learn

the IP address of the destination (real server) from the replied SYN-ACK packet, communicates directly with the real server afterwards. Proxy will parse the client/server binary protocol of MySQL/PostgreSQL and extract queries, send RW and transactional queries to the leader replica, and redirect read-only (RO) queries to RO replicas for read-intensive workloads. In addition, proxy supports pragmatic features like horizontal partitioning and connection pooling.

TcpRT collects traces from proxy and database servers only. Neither vSwitches nor SLB/NGLB nodes are collected for the following reasons. First of all, SLB and vSwitches are infrastructures shared between RDS and other cloud products, it's not appropriate to bring in any overhead to other products for RDS internal demands. Secondly, in SLB DNAT mode, or with NGLB, SLB/NGLB cannot see the complete traffic flows, so it's less meaningful to collect trace points there. And thirdly, SLB is optimized to handle a high throughput (160 Gb per second in a single machine) without the involvement of CPU, which is in the opposite direction of fine granularity tracing. Nevertheless, both vSwitches and SLB can provide coarse granularity statistic metrics about their states, it's the future work to integrate their metrics into TcpRT when performing root cause analysis.

TcpRT could also provide feedbacks to the control plane. For instance, when a TOR is diagnosed as anomaly, TcpRT can immediately ask the HA manager to switch all instances under this TOR to other AZs. If slow disks or memory is detected on a physical machine, TcpRT will command the resource manager to mark the machine as not allocatable, ask the HA manager to perform migrations, arrange the migration manager to clone and reconstruct new replicas on other machines and finally mark the machine as offline when all migration plans are completed.

2.2 Challenges

Challenges in TcpRT:

- The network environment between user applications and database processes is quite sophisticated, especially with the multi-tier data plane architecture and various optimization techniques in load balancers. It's essential to take the network into consideration when we evaluate user experience of cloud database and find the root cause of performance degradation.
- There are a large number of database instances on the cloud platform, belonging to different customers with unpredictable accessing patterns. Manually adjusting thresholds is no longer feasible to our scenario, we need self-adjustable models as the decision criteria. Shorter recovery time is a most important requirement from our customers, which depends on faster and automatic fault detection and diagnosis.
- Trace should be enabled for all instances to capture rare and ephemeral anomalies in production environment. It requires TcpRT to have little impact to database performance while collecting sufficient runtime information. The ability to process trace data in real-time is also very important. It should be scalable because the amount of data is increasing with the number of instances. In addition, TcpRT itself should be failure-resilient and each record should be processed exact-once.

Figure 2: The architecture of the comprehensive TcpRT system.

3 OVERVIEW

Figure 2 illustrates TcpRT's architecture. There is a TcpRT kernel module (see Section 4) collecting metrics of each query, like the end-to-end query latency and occurrences of network anomalous events, on both proxy and database nodes. Trace data is transferred to the Gatherer process (see Section 5.1) which is running in user space. After local aggregations, the amount of data is reduced dramatically. The results are then written to Kafka, a distributed message queue system.

TcpRT ETL job (see Section 5.2) is running on JStorm. A pool of JStorm spout threads retrieve data from all partitions of Kafka topics in parallel, grouping inputs by instance ID, so that various sources of information related to a particular instance, e.g., collected separately from proxy, leader replica and standby replica, is flowed into to a single bolt thread, where it is joined and analyzed. Time series of various metrics are generated at multiple dimensions and granularities. For example, the average latency spent in proxy node (average proxy latency, see Section 6.3) is being calculated. RDS metadata is required in ETL to perform the transformation and joining. An example scenario, where data collected from proxy node uses the VPC IP address and port as the identifier, while data collected from database node uses the physical IP address and port as the identifier, which should be all be mapped into the instance ID. Time series outputs are cached in Redis cluster for 15 minutes, after which they are expired and flushed to HybridDB.

TcpRT's automatic anomaly detection system (see Section 6.1) scans time series in Redis cluster and HybridDB periodically. It finds out a list of suspicious instances with anomalous events, and use statistical methods to decide whether it's caused by a fault of database node (see Section 6.2), proxy node (see Section 6.3) or network component (see Section 6.4).

4 TCPRT KERNEL MODULE

Well known databases usually tend to define proprietary protocols between client and server, such as MySQL[15] client/server protocol, PostgreSQL[14] frontend/backend protocol, Redis[17] serialization protocol, MongoDB[7] wire protocol. Most of them are simple request/response protocol model.

As shown in Figure 3, client sends a request to server, and server sends back a response after processing it. Both the request and the response could be segmented into multiple packets by the network

Figure 3: TCP request-process protocol.

stack. At server side, we define T_0 as the timestamp of first ACK sent from server, T_1 as the timestamp of the last ACK sent from server, T_2 as the timestamp of first data packet sent from server, and T_3 as the timestamp of last ACK received from client. Then we have the following indexes:

- Upload time = $T_1 - T_0$.
- Process time = $T_2 - T_1$.
- Download time = $T_3 - T_2$.
- Query time = $T_3 - T_0$.
- RTT time = $T_2 - T_2'$.

Upload time is approximate to the time spent to transfer the request from client to server on network. It is actually a little smaller than the real value, since it doesn't take account of the time cost for the first packet to transfer. However, the larger the request is, the more packets would be segmented into, and the more accuracy upload time would be, i.e., bulk inserts.

Process time is the time spent to process the query by server. It equals to the time cost to process the request, including both user time (spent in server application) and system time (spent in OS kernel).

Download time is the time spent to transfer the response from server to client on network.

Query time is the sum of upload time, process time and download time, which is approximate to the end-to-end latency of queries execution time experienced by client, including the time for server to process the query, and the time spent to transfer request and respond packets on network.

RTT time is the round trip time between sending a packet and receiving its ACK.

T_0 , T_1 , T_2 , T_3 are collected efficiently inside TCP congestion control kernel module, which provides callbacks for intercepting all packets sent and all ACKs received. TcpRT kernel module is highly optimized to minimize the overhead. The context for each TCP connection is piggybacked to the *tcp_sock* kernel object to save memory allocation and avoid extra locks due to sharing data structures between kernel threads. Timestamps are obtained by invoking fast *do_gettimeofday*. Unnecessary invocations to *do_gettimeofday* are all trimmed, only the first result packets sent out needs be

timestamped, and ACKs of TCP keep-alive packets which have no payload are ignored. All data structures in TcpRT kernel module are carefully aligned to avoid false-sharing to improve scalability.

A large amount of trace records collected in kernel module need be transferred to user space for further aggregation and network transmission. We choose to use a customized debugfs, a high performance in-memory filesystem. Each file in debugfs acts as a ring buffer in kernel which is also memory-mapped to user space, so that one extra memory copy operation cost can be eliminated. To avoid lock contentions, we bind an individual debugfs file to each CPU core. To decrease the frequencies of writing operations, we add a kernel buffer for each debugfs file, trace points are first appended into kernel buffer, and then batch flushed into debugfs file when the buffer is full or a timeout event is triggered.

Figure 4: TcpRT performance impact.

Figure 4 shows the impact to database performance after TcpRT is enabled. We run a sysbench with 400 client connections against MySQL. TcpRT has little overhead (within 1%) for all CPU configurations.

In TcpRT, various network events are captured via registering callbacks in *tcp_congestion_ops* object. *cwnd_event* and *pkts_acked* are two most important callbacks. When packets are sent, the *cwnd_event* callback is invoked, including sending FIN and RST packets. If there are packets remaining in the socket sending queue when a FIN packet is sent, usually it's caused by either abnormal client or broken downlink from server to client. In both callbacks, the number of TCP retransmissions and out of order packets can be detected by looking up the *tcp_sock* object.

5 TCPRT PIPELINE

5.1 TcpRT Gatherer

The TcpRT kernel module collects the statistics information of each query and generate tens of millions of records per second. We construct the TcpRT ETL pipeline to deal with these large amount of data.

Figure 5 shows the process. The TcpRT kernel module writes the raw data to debugfs, one record for each query. A local aggregation process that we call TcpRT Gatherer reads the raw data from debugfs, and writes to the file in the memory file system, i.e., */dev/shm*, after aggregating the raw data. Another process running on the same machine reads the file in the memory file system and writes them to Kafka.

5.1.1 Raw data format. Record written by the TcpRT kernel module consists of two parts. The first part contains the meta information about the query, and the second part is the metrics of the query.

Figure 5: TcpRT collection process.

The following are the fields of the first part:

(Version, Timestamp, ClientIP, ClientPort, ServerIP, ServerPort, VirtualPrivateCloudIP, VirtualPrivateCloudTunnelID)

The fields of the second part (Table 1 shows the meaning of these metrics):

(DownloadSize, RequestLatency, RoundTripDelay, NumberOfRetryPacket, TheNthOfRequest, ServerProcessTime, UploadTime, UploadSize, IsPacketReordering, MaximumSegmentSize)

Metric name	Description
DownloadSize	The number of bytes sent from server to client.
RequestLatency	The total time spent on the query in microseconds.
RoundTripDelay	Round trip time of TCP packet in millisecond.
NumberOfRetryPacket	The number of retransmission packets when transmitting packets to clients.
TheNthOfRequest	The sequence number of the request in a session.
ServerProcessTime	The time it takes for the request to be processed in microseconds.
UploadTime	Time for client sending queries to server in microseconds.
UploadSize	The number of bytes sent from client to server.
IsPacketReordering	If it is 1, packet reordering occurs during client sending queries to server, otherwise it is 0.
MaximumSegmentSize	The maximum amount of data can be received in a single TCP segment.

Table 1: The meaning of the metrics.

5.1.2 Local aggregation. Raw records are aggregated locally in TcpRT Gatherer in order to reduce the amount of data transmitted through the network and lower the load of downstream ETL job. First, all Tcp connections from the same client source IP address are aggregated, the key after aggregations becomes the tuple

(Timestamp, ClientIP, ServerIP, ServerPort, VirtualPrivateCloudIP, VirtualPrivateCloudTunnelID), in other words, the key is identified by 3-tuple (client ip, server ip, server port) instead of 4-tuple (client ip, client port, server ip, server port). Second, metrics values of all queries with the same 3-tuple in a fixed time window (every 5 seconds) are further aggregated and replaced with summary statistics like average, max, count and histogram.

In addition, we also estimate a few interesting properties about usage behavior, for example, the number of active connections, whether applications use connection pooling and so on.

5.2 TcpRT ETL Job

Figure 6: TcpRT job topology.

In this section we describe the JStorm job topology, illustrated in Figure 6.

5.2.1 Transformation. Transformation bolt threads parse the messages received by Kafka spout, extract the identifier from the message, transform the key into various aspects (instance id, server IP, etc) and post the message to downstream bolts for further processing. Metadata spout reads the attributes of all instances from the RDS metadata and broadcasts the data to all transformation bolts every 5 minutes. We create two HashMaps in every transformation bolt thread, one is used for mapping virtual IP address and port to the attributes of instance for the data collected from proxy node, the other HashMap is used for mapping physical IP address and port to the attributes of instance for the data collected from database node. Kafka spout sends the messages to multiple transformation bolt in the same JStorm worker process, the round-robin strategy is used for load balance.

5.2.2 Join and Aggregation. We have two kinds of aggregation tasks, one is to group the information related to a particular instance, the other is to group the information related to a particular machine (a proxy node or a database node). Transformation bolt sends the data of the same instance ID or the same machine IP to the same aggregation bolt.

Various sources of information related to a particular instance, e.g., collected separately from proxy node and database node, is transmitted to the same aggregation bolt thread, so that it is joined

and analyzed. The average latency spent in proxy node is calculated by subtracting the average Process time collected on proxy node to the average Query time on database node.

Information of all instances on a database node is also grouped together into the same aggregation bolt. A lot of server host related metrics are being extracted, i.e., the total number of active instances, QPS, network throughput, the number of TCP connections and so on. Host can be sorted with these properties to find the one with highest load, so that migration action could be scheduled to rebalance load between database nodes.

In aggregation bolt, we create a set of fixed time windows [2] for every key (Instance ID or Machine IP), these windows are of different time granularities (in seconds or minutes), records within corresponding period of time are accumulated in the window. Accumulation operations must have two properties: commutative and associative, e.g, keeping both numerator and denominator variables for the average operation, and maintain a histogram for the standard deviation operation, so that lately arrived data points can be correctly supplemented into previous statistical results. Every window would be flushed and emitted to the output bolts after expiration.

5.2.3 Data storage. Output bolt buffers the results for a while, when the buffer is full or it reaches a expiration time, it flushes the buffered data to an asynchronous queue. Output bolt initializes the writer thread at startup, that consume bulk of results from the queue and cached them to a Redis Cluster. The main reason of having an asynchronous queue is to tolerate temporary unavailability of the Redis Cluster service.

5.2.4 Delay and out-of-order arrival. A noteworthy point is how we solve the problem of a portion of events which arrive late and not being able to get processed within the time window. TcpRT use event-time in the whole system, which is the time event actually occurred, instead of process-time, which is the time event be processed. So if some events arrive late, it could not be aggregated into the window in time, which would cause inaccurate results. And it's impossible to hold and buffer all windows in the memory to wait for late events for a long time, which would violate the real-time nature of TcpRT system. We use a novel method called best-effort aggregation to balance between data integrity and real-time property, and guarantee the eventual consistency. Window are flushed in-time, however when some old data within an already expired window arrive, it's directly merged into the database. The correctness is ensured by the commutative and associative properties of aggregation operations in TcpRT system, by extracting data from database and merging the statistical variables of two datasets.

6 ANOMALY DETECTION ALGORITHMS

6.1 Detecting Significant Change in Time Series

When key metrics of a component are significantly higher or lower than average it's an indication that component isn't working properly, so it's possible to find anomalies by detecting significant changes in the time series metrics. Enterprise database administrators used to set threshold for each metric manually, however, this solution is not appropriate to cloud database vendors. First, thresholds are

specific to applications and may be changed as time goes by. Second, there are hundreds of thousands of DB instances on DBaaS platform, it's impossible to configure a customized threshold manually for each of them.

Figure 7: Time Series of TcpRT Metrics, MEAN, MEDIAN, SD, MAD of Their Historical Data.

By observing the statistic results of historical query latencies of a lot of instances (see Figure 9-a), we found it looks similar to the normal distribution. So initially we tried to use the control chart to model the behavior for each instance. The control chart is a statistic model based on normal distribution, which calculates the mean and the standard deviation from samples, draws a central line using the mean value, and defines the upper and lower limit as 3 standard deviations from the central line. The upper and lower limit in a control chart could be used as the thresholds. However, the value of mean and standard deviation vibrates dramatically when outliers exist. For example, there is a jitter in Figure 7-a appearing near 04:10, which causes MEAN to increase by about 5 times (Figure 7-b) and SD to increase by about dozens times (Figure 7-c) for a long time. The case is similar in Figure 7-d, a sharp drop lasting for a short time causes significant decrease of MEAN (Figure 7-e) and increase of SD (Figure 7-f). We need a different statistical model not heavily affected by outliers, Cauchy distribution meets our requirements as described below.

We find MEDIAN and MAD are more suitable than MEAN and SD and shows both of those two groups of statistical variables in above examples, in Figure 7-b, Figure 7-c, Figure 7-e, and Figure 7-f. MEDIAN fluctuates less severely than MEAN, and MAD is pretty steady. On the other hand, the shape of normal distribution looks quite similar to that of Cauchy distribution (Figure 8) when there isn't any outlier in samples. However, when it has outliers, Cauchy distribution is more stable, so finally we choose to use Cauchy distribution with MEDIAN and MAD as parameters,

Figure 8: Solid curve is a Cauchy density function with $z_0=10$ and $b=1$. The dashed curve is a Gaussian with the same peak as the Gaussian($1/\pi$) with MEAN=10 and variance= $\pi/2$.

and we use CDF of Cauchy distribution to detect rare events. Let the metric value at current time be x , if $CDF(x)$ is near the floor or the ceiling, its $PDF(x)$ will be close to zero, so it can be thought as a rare event.

We read the historical time series from database and calculate the MEDIAN and MAD, define the CDF of Cauchy Distribution as:

$$CDF(x') = \frac{\arctan \frac{(x' - MEDIAN)}{MAD}}{\pi} + \frac{1}{2} \quad (1)$$

Since the domain of $CDF(x')$ is $(-\infty, +\infty)$ while the range of metric value x is $[0, +\infty)$. We need a transform function $T(x)$ to transform x to x' so that it can be applied in the standard Cauchy distribution. Our transform function is:

$$T(x) = \begin{cases} x & x \geq m \\ \frac{2m \tan \frac{\pi(x-m)}{2m}}{\pi} + m & x < m \end{cases} \quad (2)$$

Let $x' = T(x)$. Figure 9-b shows the PDF of Cauchy distribution and normal distribution based on MEDIAN, MAD, MEAN, SD of the historical data from figure 9-a. We can see from figure 9-a that the x position of the central vertical line of the Cauchy distribution model is 1494 while the x position of central vertical line of the normal distribution model is 1738. The central line of historical data distribution is closer to that of Cauchy distribution while central line of Normal has obvious bias (figure 9-a, figure 9-b).

Finally, we need a function to determine if the incoming value for a metric is anomalous. Let function $H(x)$ be the function to judge whether metric is anomalous. We have

$$H(x) = \begin{cases} +1 & CDF(T(x)) > UPPER_LIMIT \\ 0 & LOWER_LIMIT \leq CDF(T(x)) \leq UPPER_LIMIT \\ -1 & CDF(T(x)) < LOWER_LIMIT \end{cases} \quad (3)$$

$LOWER_LIMIT$ and $UPPER_LIMIT$ are constants based on conditions of our system. In our system, $LOWER_LIMIT$ and $UPPER_LIMIT$ are 3% and 97% respectively. If $H(x)$ equals 0, there is no sharp upward/downward trend. If $H(x)$ equals +1, the metric follows sharp upward trend. If $H(x)$ equals -1, the metric follows sharp downward trend. Thus we can detect significant change by the value of $H(x)$.

6.2 Detecting Anomalous Machine

There are multiple DB instances running on a single host, that are used by different users, thus the metric of each DB instance could have its own behavior pattern. However, if there is something wrong with the underlying machine, most instances running on it would be influenced and follow the same trend. Figure 10

Figure 9: The distribution of historical data is closer to Cauchy distribution than a Normal distribution. (a). Use MEAN, MED, MEDIAN, MAD to plot the PDF of Normal distribution and Cauchy distribution (b), we found that the central vertical line of PDF of Cauchy is closer to that of distribution of historical data (a, b) while the central vertical line of Normal has obvious bias. We can use CDF of Cauchy distribution to detect anomalies (c).

Figure 10: When a machine stuck, average query latency rises while data size written to disks drops for all instances on it.

shows that when a host that is waiting on IO writes, for each instance on it, average query latency rises while data size written to disks drops.

We evaluate $H(x)$ of each instance on a machine, let s be the sum of values of $H(x)$. If $s > 0$, there will be an overall upward trend. If $s < 0$, there will be an overall downward trend. Let r be the ratio of s to the number of active DB instances on the machine. If a host is problematic, impacts on all active instances are expected, which will cause a sudden rise in $|r|$. Higher $|r|$ means higher probability that there is an anomaly event happening on the machine.

Figure 11: Distribution of r_{pt} and $|r_{qps}|$ on condition that $|r_{pt}| > 0.4$.

To determine a threshold of r , we plot the distribution of r of machines in our system. Figure 11-a shows that most points are within the range $[-0.4, 0.4]$. The MEDIAN and MAD of dataset from figure 11-a is 0 and 0.0043 respectively. Figure 11-b plots the CDF of Cauchy($x_0 = \text{MEDIAN} = 0, \gamma = \text{MAD} = 0.0043$).

If $r < -0.4$, then $\text{CDF}(T(r)) < 0.003$, which means the probability is below 0.003. If $r > 0.4$, then $1 - \text{CDF}(T(r)) < 1 - 0.997 = 0.003$, which means the probability is also below 0.003. When $|r| > 0.4$, we think the machine is abnormal and the probability is 0.006.

Let function $H_m(r)$ be the function to judge whether machine is abnormal. We have

$$H_m(r) = \begin{cases} 1 & |r| > 0.4 \\ 0 & |r| \leq 0.4 \end{cases} \quad (4)$$

If $H_m(r) = 1$, we think the machine is anomalous. If $H_m(r) = 0$, we think the machine is fine.

However, a probability of 0.006 is still not small enough because there are a large number of machines in our system. We can select multiple independent metrics, judge whether the machine is anomalous only when $H_m(r) = 1$ is true for all metrics, to improve the accuracy and lower the false positive ratio. As shown in Figure 11-c, the distribution of $|r_{qps}|$ on condition that $|r_{pt}| > 0.4$ also can be regarded to follow the half-Cauchy distribution [8], which is plotted in Figure 11-d, the probability of $|r_{qps}| > 0.4$ under the condition of $|r_{pt}| > 0.4$ is $1 - 0.959 = 0.041$. Since the probability of $|r_{pt}| > 0.4$ is 0.006, the probability that both $|r_{qps}| > 0.4$ and $|r_{pt}| > 0.4$ are met will be $0.006 \times 0.041 = 0.000246$ which is low enough.

6.3 Detecting Proxy Anomaly

Proxy is an important component which routes SQL queries to DB servers. A proxy node would typically interconnects with dozens of DB nodes within the same cluster, if a proxy node is anomalous, e.g., stops responding or timed out, performance of a large quantity of instances would be degraded. It is necessary for us to detect anomalous proxy as soon as possible, and remove it from servicing.

Figure 12: SQL process with proxy. T_1, T_5 are the request and response process time in proxy, T_3 is the DB process time, others are the time on the network.

First we should seek a metric to represent the extra latency brought by proxy. As shown in Figure 12, The Process time on proxy node equals $T_1 + T_2 + T_3 + T_4 + T_5$, and the Query time on DB node equals to $T_2 + T_3 + T_4$ (see Section 4 for definitions of Process time and Query time). So the extra cost brought by processing packets in proxy node should be the subtraction of the Process time on proxy node to the Query time on DB node, which equals $T_1 + T_5$.

The accurate value of proxy latency for each query cannot be obtained in TcpRT, since the detailed information about the latency of each query on proxy and database node is pruned after local aggregation. However, we can work out the approximate average proxy latency for each instance, by subtracting the average Process time on proxy node to the average Query time on DB node. In normal case this value should be quite small, and if it increases significantly for a large portion of instances, the proxy node could be problematic. The algorithm to detect whether the average proxy latency for a large number of instances rise at the same time on the same proxy is similar to the algorithm discussed in the above section.

The trend of variations of other metrics for a large number of instances, like upload size, download size, QPS, number of active connections can also be used to detect anomalous proxy nodes.

6.4 Detecting Network Anomaly

As shown in Figure 13-a, all nodes, including proxy nodes and database nodes, uplink to a pair of TOR switches to tolerate single point failure of TOR. High packet loss ratio in TOR would cause a large percentage of TCP packets retransmissions and result in increase of query latency and even connection failures, which would be regarded as database performance degradation by our customer. So it's important for us to identify network malfunction scenarios and find out which network device is anomalous, and resolve it, i.e, fix or replace, in a very short period. TcpRT collects the number of out-of-order packets, retransmitted packets, RTT jitters, RST packets on each TCP connection, which can be used for troubleshooting the network. We will describe how to find out the anomalous TOR pair using that information.

Figure 13: Example for detecting network anomaly.

As shown in Figure 13-b, nodes communicate with each other in our distributed architecture, for example, queries are redirected from proxy to database node, and there are replications between leader and standby replicas. We can draw a bipartite graph to represent the relationships between nodes, the vertexes are proxy nodes and database nodes that are communicating, if two nodes are communicating with each other, then an link is placed on the graph.

The link marked with dashed line means that a lot of network anomalous events (out-of-order, retransmissions, etc) are observed between the two nodes, otherwise we use a solid line instead.

We have the network topology knowledge of how machines are uplinked to different pairs of TORs, so we can convert Figure 13-b into Figure 13-c, by replacing nodes into TOR pairs they are uplinked to, it then plotted for whose vertexes are TOR pairs. Intuitively, the more dashed link connected to a pair of TOR, the more suspicious it would be. Let *count* be the number of dash edges directing to a vertex, *total* be total number of edges of a vertex, we define the empirical formula $\frac{count^{1.5}}{total}$ to be the likelihood of the TOR pair to be malfunctioning, the bigger the coefficient is, the more likely the pair of TOR is problematic. In Figure 13-c, we would deduce TOR pair2 is anomalous.

7 SCALABLE STREAM PROCESSING

Since TcpRT system collectors continuously generate enormous amount of data, thus we need to build a streaming data processing platform. We will introduce our streaming data processing platform in the following three sections.

7.1 Scale Out

The primary advantage of our streaming data processing platform is elastic scaling. We adopt the following approaches to improve the scalability of TcpRT system.

7.1.1 Loose Coupling. A complex streaming pipeline often consists of various modules which are components of business logic, e.g., in TcpRT system there is a collector, an aggregator, an anomaly detector, and so on. If all modules communicate with each other directly, they are coupling together. When upgrading a module (optimizing the algorithm of anomaly detection), upstream/downstream would be affected. Therefore, we use Kafka, to decouple modules. Then each module only needs to exchange intermediate data with Kafka that provides HA, persistent data and multiple replicas. In this way, we can stop and update modules without data loss, because data can be replayed based on Kafka messaging record offsets.

7.1.2 Shuffle Strategy Optimization. In TcpRT system, we need to aggregate collected metrics on multiple dimensions and granularities. For example, the average latency of querying on a DB instance per minute.

Fields grouping. With the former method, metrics collected from the same DB instance need to be sent to the same bolt thread for aggregation. To achieve this, all metrics from different DB instances need to be shuffled crosses all worker processes by fields grouping in JStorm. There is a drawback that most tuples will be serialized, transmitted and deserialized, which wastes time and CPU resources. A new shuffle strategy, local fields grouping, is designed to solve this problem.

Local fields grouping. As shown in Figure 14, instead of shuffling between processes, local fields grouping only shuffles data between task threads inside the same worker. Tuples are transferred internally without serialization and deserialization, which reduces consumption in CPU and network resources drastically. Though

Figure 14: Fields grouping vs local fields grouping

Figure 15: Loading dimension table.

we can compute partial aggregated results inside workers using local fields grouping, downstream reducer layer is still needed to merge partial results from different workers. However, it is noteworthy that the size of data which is aggregated by local fields grouping has decreased sharply.

7.1.3 Large Dimension Table. In TcpRT ETL, we need to join the stream data into dimension tables [6], e.g., adding the instance meta-info (instance name, DB type, etc) on collected metrics. These dimension tables updated infrequently are stored in the metadata database. Moreover, the size of dimension table is huge, which could be hundreds of megabytes.

In the former design, the whole dimension table is loaded in memory. Every bolt thread respectively reads dimension table from metadata database every 5 minutes. However, when the number of parallel bolts rise to hundreds, the scalability can not be guaranteed since metadata database can not support too many DB queries which would pull hundreds of megabytes data.

We introduce a spout thread to read dimension table from metadata database to broadcast to other bolts, shown in Figure 15(a). This design eliminates the bolts' scale-out issue — no matter how many bolts are running, there is only one spout reading metadata database.

Unfortunately, when job starts, all bolts need to wait for a single spout to read and broadcast dimension table for several minutes, which is called the cold start issue.

In order to solve this problems, we use Redis cache as our fast storage to replace memory in each bolt, as Figure 15(b) shows.

The benefit of using Redis is that we can load dimension table in advance that solves the cold start issue. Moreover, instead of copying the dimension tables to all bolts, only one copy of dimension table is needed to be synchronized, which solves the scale-out issue and reduces the probability of full GC problem.

In order to reduce the workload of Redis, we allocate a local cache on each bolt to save the dimension rows coming from Redis. To avoid blocking the pipeline, an asynchronous thread is created to continuously synchronize changes from Redis.

7.2 Low Latency

As mentioned above, TcpRT needs to limit latency within seconds for online anomaly detection.

Figure 16 illustrates detailed implementation of low latency.

Figure 16: Low latency with Redis cache.

Separate hot/cold data. Raw trace data is aggregated in different timeframes (1-hour, 1-minute or 1-second). The former two timeframes are saved in Hybrid DB to generate offline reports. The aggregated data in 1-second periods is cached in Redis and used for online anomaly detection. This hot cache data would expire in 15 minutes to prevent depletion of Redis cache. TcpRT reads the latest instance statistics from Redis and conducts correlation analysis. This approach allows us to complete diagnosis within seconds in 90% of the cases.

Handle late data, eventually consistency. The data collected in production environment is out of order. To achieve low latency, we update the latest data to Redis every second. However, data may be incomplete during the process, which may cause uncertainty in diagnosis results.

To solve this issue, we divide continuous time period into tumbling windows which are a series of fixed-sized, non-overlapping and contiguous time intervals (i.e., one hour can be divided into 60 1-minute tumbling windows). We assign stream data to corresponding windows according to event time (when data is generated). The time windows are triggered when certain conditions are met, e.g., we receive a tuple whose event time is later than the end time of a time window. If a window is triggered, we can apply user defined logics on data within this time window and then refresh all data.

To resolve the delayed data issue, we reserve extra time for the aggregated data when a time window is triggered. The aggregated data within the time window are checked with late coming data, and then updated. We refresh these aggregated data in Redis periodically until the time window expired so that TcpRT system can obtain complete aggregated data from Redis eventually.

7.3 Failure Resilient

We deploy the streaming pipeline on JStorm platform which provides several mechanisms to handle job failures, such as acker mechanism [1], backpressure [12], etc. In most cases, we can promise at-least-once semantic with these mechanisms. However, it is difficult to achieve exactly-once semantic according to CAP theory [4].

Therefore we propose a solution to solve our data consistency problem referred to Lambda [13] and Kappa [11] architecture.

While stream processing system guarantees low latency, we adopt batch system to promise strong consistency. Same as Kappa architecture, our streaming/batch system depends on Kafka and is used as a streaming queue. But it also can reserve long-time old data for batch system, that we tend to set the retention time as 24 hours.

When TcpRT system is restored from failures, it would resume jobs from latest snapshot where last consumed offset of Kafka is recorded. There might be a huge lag between current offset and latest offset since some malfunctions (like failure of physical network and storage) would last a few hours. To catch up with latest data, a long time (even 10 times of failure time) is required. Currently, since TcpRT system is latency-sensitive, we need to recover the real-time diagnosis as soon as possible after a failure. Therefore we reset offsets to consume latest data immediately, as Figure 17 shows. To avoid data loss, we create an independent repair job to replay the data within the failure time. The repair job does recompute the aggregated data to achieve eventual data consistency.

Figure 17: Failure resilient design.

8 CASE STUDIES

8.1 Troubleshooting Instance Issues

With TcpRT's Dashboard UI, we can choose a period of time to see the network topology of user's instance during the period clearly. A vertex indicates a client node, proxy node or database node. A link indicates that there are requests and responses between two nodes. In order to troubleshoot, we can click on the link or node we are interested in to see the metrics charts of it. To accelerate troubleshooting, we use the algorithm described in Section 6.1 to check the metrics of the link, and mark the link as red if there are serious issues. So we can see the metrics of the red link and the corresponding nodes first, this is very useful when the topology has a lot of link and nodes. Figure 18 shows a real case in our system. We found jitters in download time (See Figure 19), and high network retransmission rate on red links (See Figure 20). Since all of the links between the database node and different proxy nodes

Figure 18: The network topology with IP desensitized.

Figure 19: Download time of a link connected to the DB node.

Figure 20: Retransmission rate of a link connected to the DB node.

are marked as red, we concluded there was a problem with the network from the database node to proxy nodes.

8.2 Detecting Proxy Anomaly

Figure 21: Run chart of number of proxy alarms.

With TcpRT's automatic anomaly detection system, we find various kinds of anomalies in our system in time before they evolved to a disaster scenario. For important components such as proxy, we alarm operators by SMS and phone when anomaly happens. As an example, in the Feb 2017, we received many alarm messages about proxy from TcpRT's anomaly detection system. It reported that proxy's average performance suddenly slowed down. Troubleshooting with TcpRT's monitoring system, we found that there was a bug in proxy's schedule system. It caused proxy scheduling requests unfairly. After this bug has been fixed, the number of

alarm messages about proxy decreased sharply. Figure 21 shows that the number of proxy alarms per day was so large before Feb 18.

8.3 Helping our users find problems of their application

We received a report from one of our users, the application of the user failed to connect to database occasionally. We used the TcpRT system to troubleshoot the issue, by ruling out the possibility of the fault of our system, and we found that all the connection failure problems are from the same client. That the usage behavior of the client was changed after a certain time, the rate of new connections was greatly increased. We informed the user this information, and the user realized the application was upgraded at that time, and the connection pool configuration had been a mistake.

8.4 Evaluating the Health Status of System

We developed TcpRT's Events Summary System to collect anomalous events and generate a daily report about them every day. The daily report includes summary of events detected by TcpRT and the charts of machines and instances affected by these events. It also includes top-50 machines and top-50 proxy nodes with the largest number of events happened on it. Daily report system can help us understand the health status of our system conveniently. In early Dec 2016, we found the number of instances affected by serious events increases suddenly. We checked the top-50 machines from the daily report and found that most of machines were purchased at the same time. So we identified that it's faulty hardware scenario among these machines. After we replaced them with a different batch, the number of affected instances was reduced as expected.

9 CONCLUSION

The TcpRT is an instrument and diagnostic analysis system for cloud database service quality in Alibaba Cloud. TcpRT system gathers database TCP trace event transparently with negligible performance overhead, it applies scalable streaming computation to process tens of millions of TCP trace every second and uses a self-adjustable Cauchy distribution statistical model from historical data to detect anomalous events in database service automatically. TcpRT has been deployed for 3 years, since writing of this paper, in the production environment and it helps us solve online performance anomaly problem across the whole database service components.

10 ACKNOWLEDGEMENTS

We thank the anonymous reviewers, for their feedback on this paper. The TcpRT system has benefited greatly from the feedback of our many users within Alibaba. In addition, we especially thank Honghao Liu, the author of the initial version of TcpRT kernel module. We also thank Yongcheng Cai for helping us revise this paper.

REFERENCES

- [1] 2015. Guaranteeing Message Processing. (2015). <http://storm.apache.org/releases/1.1.0/Guaranteeing-message-processing.html>

- [2] Tyler Akidau, Robert Bradshaw, Craig Chambers, Slava Chernyak, Rafael J Fernández-Moctezuma, Reuven Lax, Sam McVeety, Daniel Mills, Frances Perry, Eric Schmidt, et al. 2015. The dataflow model: a practical approach to balancing correctness, latency, and cost in massive-scale, unbounded, out-of-order data processing. *Proceedings of the VLDB Endowment* 8, 12 (2015), 1792–1803.
- [3] Alibaba. 2014. JStorm: a distributed and fault-tolerant realtime computation system. <https://github.com/alibaba/jstorm/wiki>. (2014).
- [4] Eric A Brewer. 2000. Towards robust distributed systems. In *PODC*, Vol. 7.
- [5] Wei Cao, Feng Yu, and Jiasen Xie. 2014. Realization of the Low Cost and High Performance MySQL Cloud Database. *Proc. VLDB Endow.* 7, 13 (Aug. 2014), 1742–1747. <https://doi.org/10.14778/27333004.2733077>
- [6] Surajit Chaudhuri and Umeshwar Dayal. 1997. An overview of data warehousing and OLAP technology. *ACM Sigmod record* 26, 1 (1997), 65–74.
- [7] Kristina Chodorow. 2013. *MongoDB: The Definitive Guide: Powerful and Scalable Data Storage*. " O'Reilly Media, Inc."
- [8] Gauss M Cordeiro and Artur J Lemonte. 2011. The beta-half-Cauchy distribution. *Journal of Probability and Statistics* 2011 (2011).
- [9] Jonathan Kaldor, Jonathan Mace, Michał Bejda, Edison Gao, Wiktor Kuropatwa, Joe O'Neill, Kian Win Ong, Bill Schaller, Pingjia Shan, Brendan Viscomi, et al. 2017. Canopy: An End-to-End Performance Tracing And Analysis System. *SOSP '17* (2017), 34–50. <https://doi.org/10.1145/3132747.3132749>
- [10] Donald Kossmann, Tim Kraska, and Simon Loesing. 2010. An Evaluation of Alternative Architectures for Transaction Processing in the Cloud. *SIGMOD '10* (2010), 579–590. <https://doi.org/10.1145/1807167.1807231>
- [11] Jay Kreps. 2014. Questioning the Lambda Architecture. (2014). <https://www.oreil.ly.com/ideas/questioning-the-lambda-architecture>
- [12] Sanjeev Kulkarni, Nikunj Bhagat, Maosong Fu, Vikas Kedigehalli, Christopher Kellogg, Sailesh Mittal, Jignesh M Patel, Karthik Ramasamy, and Siddarth Taneja. 2015. Twitter heron: Stream processing at scale. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. ACM, 239–250.
- [13] Nathan Marz and James Warren. 2015. *Big Data: Principles and best practices of scalable realtime data systems*. Manning Publications Co.
- [14] Bruce Momjian. 2001. *PostgreSQL: introduction and concepts*. Vol. 192. Addison-Wesley New York.
- [15] AB MySQL. 2001. MySQL. (2001).
- [16] Benjamin H Sigelman, Luiz Andre Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. 2010. *Dapper, a large-scale distributed systems tracing infrastructure*. Technical Report. Technical report, Google, Inc.
- [17] Jeremy Zawodny. 2009. Redis: Lightweight key/value store that goes the extra mile. *Linux Magazine* 79 (2009).
- [18] Yan Zhang and Nirwan Ansari. 2013. On architecture design, congestion notification, TCP incast and power consumption in data centers. *IEEE Communications Surveys & Tutorials* 15, 1 (2013), 39–64.