Zero Downtime Release: Disruption-free Load Balancing of a Multi-Billion User Website

Usama Naseer*
Brown University

Luca Niccolini
Facebook, Inc.

Udip Pant
Facebook, Inc.

Alan Frindell
Facebook, Inc.

Ranjeeth Dasineni
Facebook, Inc.

Theophilus A. Benson
Brown University

ABSTRACT

Modern network infrastructure has evolved into a complex organism to satisfy the performance and availability requirements for the billions of users. Frequent releases such as code upgrades, bug fixes and security updates have become a norm. Millions of globally distributed infrastructure components including servers and load-balancers are restarted frequently from multiple times per-day to per-week. However, every release brings possibilities of disruptions as it can result in reduced cluster capacity, disturb intricate interaction of the components operating at large scales and disrupt the end-users by terminating their connections. The challenge is further complicated by the scale and heterogeneity of supported services and protocols.

In this paper, we leverage different components of the end-to-end networking infrastructure to prevent or mask any disruptions in face of releases. *Zero Downtime Release* is a collection of mechanisms used at Facebook to shield the end-users from any disruptions, preserve the cluster capacity and robustness of the infrastructure when updates are released globally. Our evaluation shows that these mechanisms prevent any significant cluster capacity degradation when a considerable number of productions servers and proxies are restarted and minimizes the disruption for different services (notably TCP, HTTP and publish/subscribe).

CCS CONCEPTS

• **Networks** → **Network management**; *Network protocol design.*

KEYWORDS

Update releases, Load-balancing, Reliable networks.

ACM Reference Format:
Usama Naseer, Luca Niccolini, Udip Pant, Alan Frindell, Ranjeeth Dasineni, and Theophilus A. Benson. 2020. Zero Downtime Release: Disruption-free Load Balancing of a Multi-Billion User Website. In *Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication (SIGCOMM '20), August 10–14, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3387514.3405885

*Work done while at Facebook, Inc.

SIGCOMM '20, August 10–14, 2020, Virtual Event, USA

ACM ISBN 978-1-4503-7955-7/20/08.
https://doi.org/10.1145/3387514.3405885

1 INTRODUCTION

Online service providers (OSP), e.g., Facebook, Google, Amazon, deploy massively complex code-bases on large sprawling infrastructures to deliver rich web services to *billions of users* at a high quality of experience. These code-bases are constantly being modified to introduce bug fixes, performance optimizations, security patches, new functionality, amongst a host of other reasons. Recent studies from Facebook [50, 51] show that each day tens of thousands of commits are pushed to tens of thousands of machines across the globe. In fact, the number of web-tier releases increased from once per week in 2007 to tens of times a day in 2016, each comprising of 1000s of code commits [20, 21].

At the scale of multi billion users and millions of machines, code-update and release techniques must be swift while simultaneously incurring zero downtime. Today, the state of the art approach for deploying these code changes requires draining connections from servers with the old code and incrementally restarting the servers to introduce the new code [17, 28, 52]. This technique can have a host of undesirable consequences from lowering aggregate server capacity and incurring CPU overheads to disrupting and degrading end user experience. At the scale of billions of connections, restarting connections is disastrous for the ISP, end-user, and the OSP [11, 18]. The process of connection-restart incurs a number of handshakes (e.g., TLS and TCP) which we show (in Section 2.5) consumes as much as 20% of the OSP's CPU. Additionally, the flood of new connections triggers wireless protocol signaling at the cellular base-station which simultaneously drains a mobile phone's batteries and can overwhelm the cellular provider's infrastructure. Finally, we observed that during the restarts, users can experience QoE degradation and disruptions in the form of errors (e.g., HTTP 500 error) and slower page loads times (i.e., due to retries over high-RTT WAN).

Motivated by the high code volatility and the potential disruption arising from code deployment, many online service providers have turned their attention to designing practical and light-weight approaches for transparently deploying code in a disruption free manner, i.e., deploying code while ensuring zero downtime. The design of such a disruption free update mechanism for large providers is challenge by the following characteristics which are unique to the scale at which we operate: first, large providers employ a large range of protocols and services, thus, the update mechanisms must be general and robust to different services. For example, we run services over both HTTP and MQTT (a publish-subscribe protocol [45]) which have distinctly different tolerance and state requirements. Second, while many applications are stateless, a non-trivial set of applications are stateful, thus the update mechanisms must be able to seamlessly migrate or transparently recreate this state at the new server. For

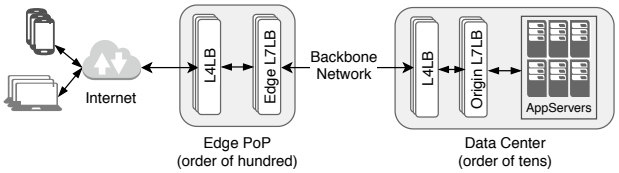


Figure (1) End-to-end Infrastructure

example, a non-trivial number of connection are long-lived and often transfer large objects, failing to migrate state can significantly impact end-user performance. Third, due to energy reasons and application-specific requirements, a subset of servers are resource-constrained (e.g., cache priming [12, 29] for HHVM servers [5] consumes most available memory), which prevents us from running both the new code and old code on the same server, thus preventing us from leveraging kernel mechanisms to migrate connections.

Our framework builds on several unique characteristics shared by the infrastructure of online service providers such as Facebook and Google. First, the end-to-end infrastructure is owned and administered by the provider which implies that updates in a specific tier, can leverage a downstream tier to shield the end-user from disruptions, e.g., the application tier can leverage the reverse proxy tier. Second, while application state is hard to extract and isolate, for long lived connections with large objects, we can recreate the state by identifying the partial requests at the old server and replaying them to the new server. Together these insights allow us to transparently migrate from old to new code while restarting servers without exposing the loss of state or server termination to the end-user.

In our framework, an update is achieved by signaling the upstream tier to handle connection migration and by indirectly externalizing user-specific state and redirecting the state to an upstream tier. The upstream tier redirects existing connections and the externalizes the state to the new servers (or servers with the new code). Additionally, zero downtime for a restarting L7LB is achieved by handing-over traffic to side-car (instance with the new code) on the same machine. To this end our framework consists of two mechanisms: a technique for signaling and orchestrating connection hand-off via an upstream tier, a method for detecting and externalizing state, and enhancements to pre-existing hand-off kernel-based mechanisms to enable them to scale to billions of users.

This framework has been deployed at Facebook for several years and has helped to sustain an aggressive release schedule on a daily basis. While comparing our framework to previously used release methodologies, we observed that our framework provided the following benefits: (i) we reduced the release times to 25 and 90 minutes, for the *App. Server* tier and the L7LB tiers respectively, (ii) we were able to increase the effective L7LB CPU capacity by 15-20% , and (iii) prevent millions of error codes from being propagated to the end-user.

2 BACKGROUND AND MOTIVATION

In this section, we introduce Facebook's end-to-end web serving infrastructure and present motivational measurements from the production clusters.

2.1 Traffic Infrastructure

Figure 1 provides an overview of Facebook's geographically distributed multi-tiered traffic infrastructure, comprising of *DataCenter* (order of tens) and *Edge PoPs* (Point-of-Presence, order of hundreds). At each infrastructure tier, Facebook uses software load-balancers (LB) to efficiently handle diverse user workload requirements and QoE. Additionally, at the origin data centers, there are also application servers in addition to the LBs.

- **L4LB (Layer4LB):** Facebook uses *Katran*, a transparent, XDP-based [7], L4LB layer that serves as a bridge in between the network routers and L7LB (proxies). Routers use ECMP [30] to evenly distribute packets across the L4LB layer, which in turn uses consistent hashing [7, 26] to load-balance across the fleet of L7LBs.
- **L7LB (Layer7LB):** For L7 load-balancing, Facebook uses *Proxygen*, an in-house proxy with responsibilities encompassing beyond those a typical traditional L7LB shoulders. Operating in different modes, it serves as the reverse proxy for load-balancing, forward proxy for outbound requests and HTTP server. *Proxygen* is the heart of traffic management at Facebook, supporting multiple transport protocols (TCP, UDP), application protocols (HTTP/1.1, HTTP/2, QUIC, publish/subscribe [45] etc.), serving cached content for CDN, maintaining security (TLS etc.), health-checking and monitoring upstream app. servers etc.
- ***App. Server* tier:** Resides in the *DataCenter* and ranges from web (HHVM, django, custom apps. built leveraging the *Proxygen* HTTP server library [1, 5, 53]) to special-purpose servers (e.g., Publish/Subscribe brokers [45]). *HHVM* servers (our focus in application tier) are a general purpose application server for HACK [54], with workloads dominated by short-lived API requests. However, they also service long-lived workloads (e.g., HTTP POST uploads).

2.2 Life of a Request

In this section we present a detailed view of how user requests are processed by the various tiers of our infrastructure and in doing so we highlight different application workflows and how they are treated.

(1) *Edge PoP* serves as the gateway into our infrastructure for a user's request and connections (TCP and TLS). These user requests/connections are terminated by the *Edge Proxygen*.

(2) *Edge Proxygen* processes each request and, if the request cannot be serviced at the *Edge*, it forwards the request to the upstream *Origin DataCenter*. Otherwise, for cache-able content (e.g., web, videos etc.) it responds to the user using *Direct Server Return* [7].

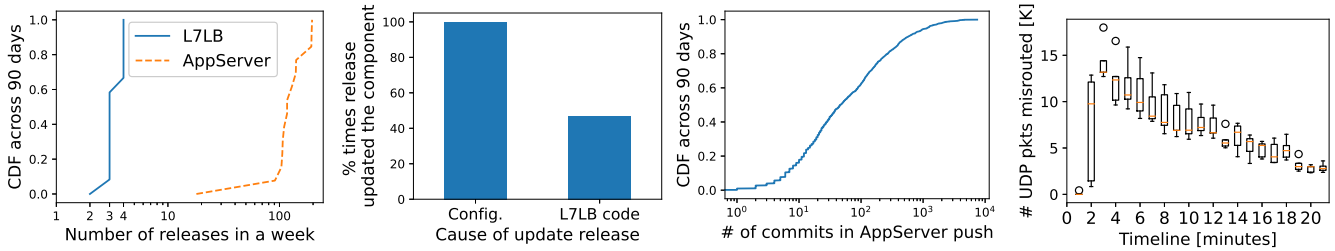
(3) *Edge* and *Origin* maintains long-lived HTTP/2 connections over which user requests and MQTT connections are forwarded.

(4) *Origin Proxygen* forwards the request to the corresponding *App. Server* based on the request's context (e.g., web requests to *HHVM*, django servers while persistent pub/sub connections to their respective MQTT broker back-ends).

In this paper, we focus on restarts of *Proxygen* (at *Edge* and *Origin*) and *HHVM App. Server* (at *DataCenter*), and focus on the traffic for cache-able, uncache-able, and MQTT-backed content. Our goal is to design a framework that shields transport (TCP and UDP) and application protocols (HTTP and MQTT) from disruption, while still maintaining reasonable update-speeds and zero downtime. This work does not raise any ethical issues.

2.3 Release Updates

Traditionally, operators rely on over-provisioning the deployments and incrementally release updates to subset of machines in batches. Each restarting instance enters a ***draining mode*** during which it receives no new connections (by failing health-checks from *Katran*



(a) # of releases **(b) Root of L7LB restarts** **(c) Root of app. tier restarts** **(d) UDP mis-routing**

Figure (2)

to remove the instance from the routing ring). This phase stays active for the ***draining period*** [14, 15, 28], the time duration deemed enough for existing connections to organically terminate. Once draining period concludes, the existing connections are terminated and the instance is restarted and the new code kicks-in.

2.4 Motivating Frequent Restarts

To swiftly address security concerns and adapt to evolving user expectations, frequent code releases have become the norm [21, 31, 50, 51], not the exception. In Figure 2a, we present the number of global roll-outs per week, over a period of 3 months for 10 Facebook's *Edge* and *DataCenter* clusters.

L7LB: Globally, at the L7LB tier, we observe on average three or more releases per week. In Figure 2b, we analyze the root-cause of these releases and observe that the dominants factors are binary (i.e., code) and configuration updates. We note that unlike other organizations, where configuration changes might not necessitate a release, Facebook requires restarting the instances for configuration update. This is an artifact of system design and, since *Zero Downtime Release*-powered restarts do not results in disruptions, it removes the complexity of maintaining different code paths, i.e., one for robust restarts for binary updates and another for configuration-related changes. Binary updates (due to code changes) always necessitate a restart and account for ∼47% of the releases, translating to multiple times a week.

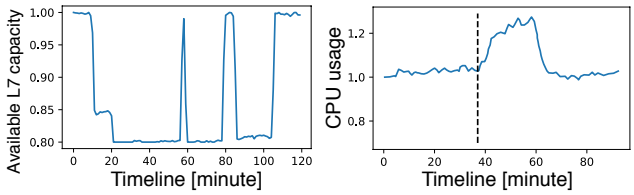
***App. Server*:** At the *App. Server* tier (Figure 2a), we observe that, at the median, updates are released as frequently as 100 times a week [50, 51]. We also observed that each update contains any where from 10 to 1000 distinct code commits (Figure 2c) and such high degree of code evolution necessitates frequent restarts. Conversations with Facebook developers identified that the constant code changes to the app. tier is a function a cultural adoption of the "Continuous Release" philosophy. Although the *App. Server* tier evolves at a much higher frequency, the impact of their restarts can be mitigated as L7LBs terminate user connections and can shield the users from the *App. Server* restarts. However, due to the stateful nature of the *App. Server* tier and the high frequency of code updates (and thus restarts), some users are bound to receive errors codes (e.g., HTTP 500) and timeouts.

2.5 Implications of Restarts

At Facebook scale, implications and consequences of frequent releases can be subtle and far-reaching. The "disruption" induced by a release is measured along multiple dimensions, ranging from increase in resource usage at CSP-end to a higher number of failed user requests. Specifically at Facebook, any irregular increase in the number of HTTP errors (e.g., 500 code), proxy errors (e.g., timeouts), connection terminations (e.g., TCP RSTs) and QoE degradation (e.g., increased tail latency) quantify the extent of release-related disruptions. Next, we discuss the direct and indirect consequences of a release.

- **Reduced Cluster Capacity:** Intuitively, during a rolling update, servers with old code will stop serving traffic and this will reduce the cluster's effective capacity. An unexpected consequence of reduced capacity is increased contention and higher tail latencies. To illustrate this point, in Figure 3a, we plot the capacity for an *Edge* cluster during a release. From this figure, we observe that during the update, the cluster is persistently at less than 85% capacity which corresponds to the rolling update batches which are either 15% or 20% of the total number of machines. Minutes 57 and 80-83 correspond to time gap when one batch finished and the other started. In a complementary experiment, we analyzed the tail latency and observed significant increase due to a 10% reduced cluster capacity .
- **Increased CPU Utilization:** During a server restart, the application and protocol states maintained by the server will be lost. Clients reconnecting to Facebook's infrastructure will need to renegotiate this application and protocol state (e.g., TCP, TLS state). In Figure 3b, we plot the CPU utilization at the *App. Server*-tier when clients reconnect. We observed that when 10% of *Origin Proxygen* restart, the app. cluster uses 20% of CPU cycles to rebuild state. This overhead mirrors observations made by other online service providers [11, 18].
- **Disrupted ISP Operations:** At the scale of our deployment, a restart of billions of connections can also put stress on the underlying ISP networking infrastructure, especially the cellular towers. On multiple instances, ISPs have explicitly complained about the abrupt re-connection behavior and the resulting congestion on the last-mile cellular connections.
- **Disrupted end-user quality of experience:** Despite efforts to drain connections, restarts at the *Proxygen* or *App. Server* tiers lead to disruptions for users with long-lived connections (e.g., MQTT) which outlive the draining period. These disruptions manifest themselves in multiple ways ranging from explicit disruption (e.g., HTTP 500 error codes) to degraded user experience (e.g., retries). At our scale, we observe that at the tail (i.e., p99.9) most requests are sufficiently large enough to outlive the draining period. In such situations, users need to re-upload or restarts session interactions.

The state of the art for performing updates, i.e., draining and rolling updates, may be suitable for a large number of medium sized online service providers. However, at our scale, the collateral damage of applying these techniques extends beyond our infrastructure and users, to the underlying Internet and cellular fabric.



(a) Cluster capacity. **(b) CPU spike.**

Figure (3) Impact of release.

3 DESIGN CHOICES

With the growing adoption of mono-repos [25] and micro-services [47], the need for a zero-disruption code release mechanism is increasingly becoming important both at Facebook and at other companies. In this section, we elaborate on emerging production techniques and discuss how these technique fail to operate at our scale.

3.1 Ideal Design (*PerfectReload*)

Abstractly, there are three ways to seamless update code.

• Language Support (Option-1): The first is to use a language with built-in support for headless updates such as Erlang [13]. Unfortunately, this approach is not supported by most common languages. In fact, only a trivially small number of our services are written in such a language. Orthogonally, the effort required to rewrite our entire fleet of services in such a language is practically unrealistic.

• Kernel Support (Option-2): A more promising alternative is to leverage the support from operating systems for seamlessly migrating connections between two processes – from the one running old code to the one running new code. However, this has two main drawbacks: First, kernel techniques like *SO_REUSEPORT* do not ensure consistent routing of packets. During such migration of a socket, there is a temporary moment before the old process relinquishes the control over it when both processes will have control over the socket and packets arriving at it. While temporary, such ambiguous moments can have a significant impact at our scale. To highlight this problem, in Figure 2d, we examine the number of misrouted packets during a socket handover. These misrouted packets illicit errors code from the server which will propagate to the end-users. While there are solutions to enforce consistency, e.g., *CMSG*, *SCM_RIGHTS* [39] and *dup()* [40], they are still limited for UDP. Second, these techniques are not scalable owing to the hundreds of thousands of connections at scale per instance. Additionally, a subset of the socket state e.g., TLS, may not be copied across process boundaries because of the security implications of transferring sensitive data.

Ignoring the scalability and consistency issues, we note that two key issues remain un-addressed. First, for long lived connections, we may need to wait for an infinite amount of time for these connections to organically drain out, which is impractical. Second, application-specific states are not transfered by *SO_REUSEPORT*, only new socket instances are added to the same socket family. Terminating these long lived connections or ignoring application state will lead to user facing errors. Thus, we need a general solution for addressing application state and for explicitly migrating long-lived connections.

• Protocol and Kernel Support (Option-3): We can address scalability issues by only migrating listening sockets across application processes and allowing the old instance to drain gracefully. For long-lived connections, we can leverage a third party server such as an upstream component to help drain and coordinate migrations of active connections. For example, during the update of *App. Servers*, an *App. Servers* can signal the *Proxygen* to migrate connections by terminating active connections and setting up new connections. However, for this approach to be disruption free, it requires the connection's protocol to support graceful shutdown semantics (for e.g. HTTP/2's GoAways) which not supported by a significant subset of our connections such as HTTP/1.1 or MQTT. Moreover, existing APIs and mechanisms in the kernel are inadequate to achieve such migration of sockets properly without causing disruptions for UDP based applications.

The ideal disruption-free release system is expected to:

(1) **Generic:** Generalizes across the plethora of services and protocols. As demonstrated with Options-3 and Options-1, specialized solutions limited to specific languages or protocol do not work at large providers where multiple protocols and languages need to be supported.
(2) **Preserve State:** Should either reserve or externalize states to prevent the overhead of rebuilding them after restarts.
(3) **Tackle long-lived connections:** Should be able to prevent long-lived connection disruptions as draining period is not enough for them to gracefully conclude.

4 *ZERO DOWNTIME RELEASE*

In this section, we present the design and implementation details for framework to achieve *Zero Downtime Release*. Our framework extends on *option-3* and introduces primitives to orchestrate scheduling, state externalization, and design enhancements to enable pre-existing operating system to scale to our requirements. In particular, our framework introduces three update mechanisms, and are composed to address the unique requirements of the different tiers in the end-to-end infrastructure. Next, we elaborate on each of these three update/release mechanisms and illustrate how our primitives are used and in particular how the unique challenges are addressed. The key idea behind the techniques is to leverage *Proxygen*, either its building blocks (e.g., sockets) or its position in end-to-end hierarchy, to prevent or mask any possible disruption from end-users.

4.1 Socket Takeover

Socket Takeover enables *Zero Downtime Restarts* for *Proxygen* by spinning up an updated instance in parallel that takes-over the listening sockets, whereas the old instance goes into graceful draining phase. The new instance assumes the responsibility of serving the new connections and responding to health-check probes from the L4LB *Katran*. Old connections are served by the older instance until the end of draining period, after which other mechanism (e.g., *Downstream Connection Reuse*) kicks in.

Implementation: We work-around the technical limitations discussed in (§ 4) by passing file descriptors (FDs) for each listening socket from the older process to the new one so that the listening sockets for each service addresses are never closed (and hence no downtime). As we pass an open FD from the old process to the newly spun one, both the passing and the receiving process share the same file table entry for the listening socket and handle separate accepted connections on which they serve connection level transactions. We leverage the following Linux kernel features to achieve this:

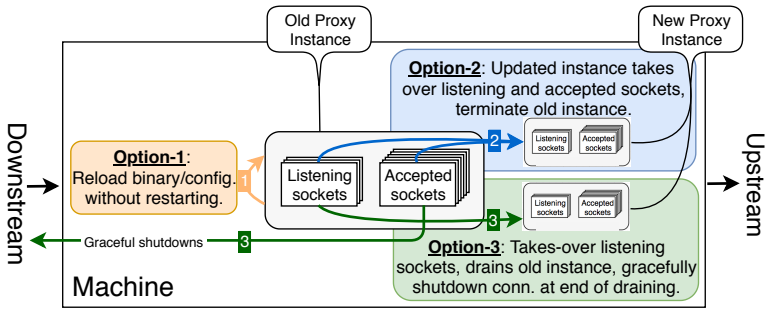


Figure (4) Design choices for PerfectReload

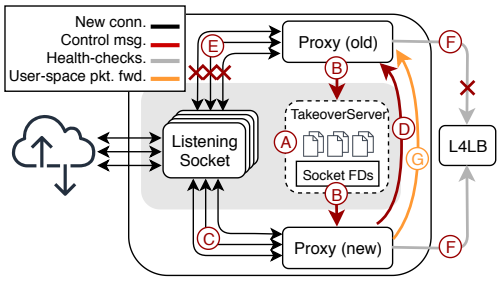


Figure (5) Socket Takeover

(1) **CMSG:** A feature in *sendmsg()* [39, 42] allows sending control messages between local processes (commonly referred to as ancillary data). During the restart of L7LB processes, we use this mechanism to send the set of FDs for all active listening sockets for each VIP (Virtual IP of service) from the active instance to the newly-spun instance. This data is exchanged using *sendmsg(2)* and *recvmsg(2)* [39, 41, 42] over a UNIX domain socket.

(2) **SCM_RIGHTS:** We set this option to send open FDs with the data portion containing an integer array of the open FDs. On the receiving side, these FDs behave as though they have been created with *dup(2)* [40].

Support for UDP protocol: Transferring a listening socket's FDs to a new process is a well-known technique [35, 36, 38, 43] has been added to other proxies like (HAProxy [3]) in 2018 [4] and more recently to Envoy [16]. Although the motivation is same, *Proxygen*'s *Socket Takeover* is more comprehensive as it supports and addresses potential disruptions for multiple transport protocols (TCP and UDP).

Although UDP is connection-less, many applications and protocols built on top of UDP (QUIC, webRTC etc.) do maintain states for each *flow*. For TCP connections, the separation of listening and accepted sockets by Linux Kernel lessens the burden to maintain consistency from the user-space application. On the other hand, UDP has no such in-built support. Typically, a consistent routing of UDP packets to a socket is achieved via application of hashing function on the source and the destination address for each packet. When *SO_REUSEPORT* socket option is used for an UDP address (VIP), Kernel's internal representation of the socket ring associated with respective UDP VIP is in flux during a release – new process binds to same address and new entries are added to socket ring, while the old process shutdowns and gets its entries purged from the socket ring. This flux breaks the consistency in picking up a socket for the same 4-tuple combination. This significantly increases the likelihood of UDP packets being misrouted to a different process that does not have state for that flow (Figure 2d).

Owing to mis-routing of UDP packets, a typical *SO_REUSEPORT*-less architecture uses a thread dedicated to accepting new connection that hands off newly accepted connections to worker threads. UDP being datagram centric and without any notion of packet type such as SYN for TCP, the kernel cannot discriminate between a packet for a new connection vs an existing one. The approach of using one thread to accept all the packet cannot scale for high loads since the responsibilities of maintaining states and multiplexing of UDP packets must now happen in the application layer, and such thread becomes a bottleneck.

To circumvent the scaling issues, we use SO_REUSEPORT option with multiple server threads accepting and processing the packets independently. To solve the mis-routing issue, we extended the *Socket Takeover* to pass FDs for all UDP VIP sockets. This essentially is equivalent of calling *dup()* [40] on an existing socket FD and thus avoids creation of a new FD for this socket. In other words, the socket ring for this VIP within kernel remains unchanged with each process restart. The newly spun process can thus resume processing packets from where the old process left off.

However, one problem still remains un-addressed. All the packets are now routed to the new process, including the ones for connections owned by the old (existing) process. For applications that require and maintain states for each UDP based flow (e.g., QUIC), the new process employs user-space routing and forwards packets to the old process through a pre-configured host local addresses. Decisions for user-space routing of packets are be made based on information present in each UDP packet, such as *connection ID* [33] that is present in each QUIC packet header. In our implementation this mechanism effectively eliminated all the cases of mis-routing of UDP packets while still being able to leverage multiple threads for better scalability.

Workflow: (Figure 5) An existing *Proxygen* process that is serving traffic has already bound and called accept() on socket FDs per VIP. In step Ⓐ, the old *Proxygen* instance spawns a *Socket Takeover* server that is bound to a pre-specified path and the new *Proxygen* process starts and connects to it. In step Ⓑ, the *Socket Takeover* server then sends the list of FDs it has bound, to the new process via the UNIX domain socket with sendmsg() and CMSG. The new *Proxygen* process listens to the VIP corresponding to the FDs (step Ⓒ). It then sends confirmation to the old server to start draining the existing connections (step Ⓓ). Upon receiving the confirmation from the new process, the old process stops handling new connections and starts draining existing connections (step Ⓔ). In step Ⓕ, the new instance takes over the responsibility of responding to health-check probes from the L4LB layer (*Katran*). Step Ⓖ stays active until the end of draining period and UDP packets belonging to the older process are routed in user-space to the respective process.

View from L4 as L7 restarts: An L4LB (*Katran*) that sits in front of the L7 proxy and continuously health-checks (HC) each L7LB is agnostic of updates in L7LB servers because of the *zero downtime restart* mechanism. We can thus isolate the L7 restarts only to that layer. This not only simplifies our regular release process but also help in reliability of the overall system since the blast radius of a buggy release is largely confined to one layer where mitigation (or rollbacks) can be applied swiftly.

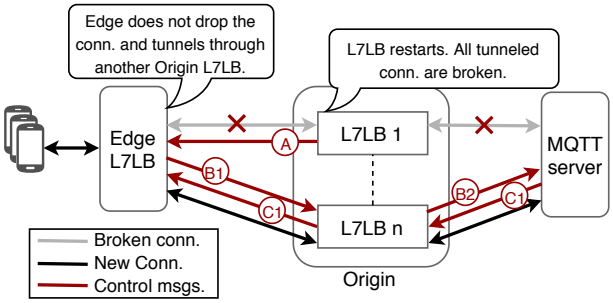


Figure (6) Downstream Connection Reuse

Connections between *Edge* and *Origin*: *Proxygen* in *Edge* and *Origin* maintain long-lived HTTP/2 connections (§ 2) between them. Leveraging *GOAWAY*, they are gracefully terminated over the draining period and the two establish new connections to tunnel user connections and requests without end-user disruption.

Caveats: *Socket Takeover* process is not free-of-cost and incurs CPU and memory overhead, as the two *Proxygen* instances run parallel on the same machine. Although the machine stays available to serve new connections, there's (often insignificant) diminished CPU capacity for the draining duration (§ 6.3).

4.2 Downstream Connection Reuse

As described earlier, MQTT is used to keep persistent connections with billions of users and the protocol requires underlying transport session to be always available (e.g., for live notifications). As a result, MQTT clients periodically exchange ping and initiate new connections as soon as transport layer sessions are broken. MQTT does not have a built-in disruption avoidance support in case of *Proxygen* restarts and relies on client re-connects. Given the lack of *GOAWAY*-like support in the protocol, the edge only has the options of either waiting for the clients to organically close the connection or forcefully close it and rely on client-side reconnects.

A key property for MQTT connections is that the intermediary LBs only relay packets between pub/sub clients and their respective back-ends (pub/sub brokers) and as long as the two are connected, it does not matter which *Proxygen* relayed the packets. MQTT connections are tunneled through *Edge* to *Origin* to MQTT back-ends over HTTP/2 (§ 2). Any restarts at *Origin* are transparent to the end-users as their connections with the *Edge* remains undisturbed. If we can reconnect *Edge Proxygen* and MQTT servers through another *Origin Proxygen*, while the instance in question is restarting, end users do not need to reconnect at all. This property makes *Origin Proxygen* "stateless" w.r.t MQTT tunnels as the *Origin* only relays the packets. Leveraging this statelessness, disruptions can be avoided for these protocols that do not natively support it – a mechanism called *Downstream Connection Reuse* (DCR).

Each end-user has a globally unique ID (*user-id*) and is used to route the messages from *Origin* to the right MQTT back-end (having the context of the end-user connection). Consistent hashing [7, 26] is used to keep these mappings consistent at scale and, in case of a restart, another *Proxygen* instance can take-over the relaying responsibility without any end-user involvement and disruption (at back-end or user side).

Workflow: Figure 6 presents the details of the *Downstream Connection Reuse* transactions. When an *Origin Proxygen* instance is restarting, it send a *reconnect_solicitation* message to downstream *Edge* LB step Ⓐ to signal the restart. Instead of dropping the connection, *Edge* LB sends out *re_connect* (contains *user-id*) to *Origin*, where another healthy LB is used to relay the packets to corresponding back-end (located through *user-id*) (steps Ⓑ1, Ⓑ2). MQTT back-end looks for the end-user's connection context and accepts *re_connect* (if one exists) and sends back *connect_ack* (steps Ⓒ1, Ⓒ2). Otherwise, *re_connect* is refused (by sending *connect_refuse*), *Edge* drops the connection and the end-user will re-initiate the connection in the normal way.

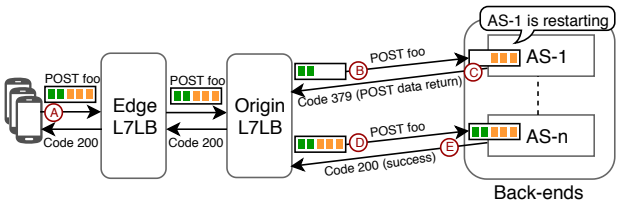


Figure (7) Partial Post Replay

Caveats: Restarts at *Origin* is the ideal scenario for DCR as *Edge* is the next downstream and can thus keep the restarts transparent to users. For a restart at the *Edge*, the same workflow can be used with end-users, especially mobile clients, to minimize disruptions (by pro-actively re-connecting). Additionally, DCR is possible due to the design choice of tunneling MQTT over HTTP/2, that has in-built graceful shutdown (*GOAWAYs*).

4.3 Partial Post Replay

An *App. Server* restart result in disruption for HTTP requests. Due to their brief draining periods (10-15s), long uploads (POST requests) pose a specific pain point and are responsible for the most disruptions during restarts.

A user's POST request makes its way to the *App. Server* through the *Edge* and *Origin Proxygen*. When the app. server restarts, it can react in multiple ways: (i) The *App. Server* fails the request, responds with a 500 code and the HTTP error code makes it way to user. The end-user observes the disruption in form of "Internal Server Error" and the request gets terminated (disrupted). (ii) The *App. Server* can fail the request with 307 code (Temporary Redirect), leading to a request re-try from scratch over high-RTT WAN (performance overhead). (iii) The 307 redirect can be improved by buffering the POST request data at the *Origin* L7LB. Instead of propagating the error (500 or 307 retry) to the user, the *Origin* L7LB can retry the buffered request to another healthy *App. Server*. The massive overhead of buffering every POST request until completion makes this option impractical.

In light of deficiencies of alternate effective techniques, *Partial Post Replay* [27] leverages the existence of a downstream *Proxygen* and the request data at restarting app. server to *hand-over* the incomplete, in-process requests to another *App. Server*. A new HTTP code (379) is introduced to indicate a restart to the downstream *Proxygen*.

Workflow: Figure 7 presents a high-level view of the workflow. A user makes a POST request (Ⓐ) which is forwarded to an *App. Server* (Ⓑ). When the *App. Server* (AS in Figure 7) restarts, it responds to any unprocessed requests with incomplete bodies by sending a 379 status code and the partially received POST request data, back to downstream *Origin Proxygen* (Ⓒ). For HTTP/1.1, the status message is set to "Partial POST Replay". The *Proxygen* instance does not send the

379 status code to the end-user and instead builds the original request and replays it to another healthy *App. Server* Ⓓ. When the request gets completed, the success code is returned back to the end-user Ⓔ.

Caveats: There is a bandwidth overhead associated with replaying the request data — high bandwidth connections, existing between *Origin Proxygen* and *App. Server* in *DataCenter*, are required to make *Partial Post Replay* viable. Additionally, since end-user applications are not expected to understand code 379, it should never be sent back to end-user. In case when intermediary cannot replay request to another server, the requests should be failed with standard 500 code.

4.4 Applicability Considerations

Next, we discuss the applicability of the three mechanisms to the different tiers. As mentioned in § 2, the tiers differ in their functionality and operational capacity, and can also have specific operational limitations, e.g., *App. Server* tier is restarted hundreds of times a week and requires draining period in order of tens of seconds. Such operational aspects decide the applicability of the three mechanism.

Whereas *Socket Takeover* is used at every *Proxygen* instance in the end-to-end architecture, *Socket Takeover* is not used for the *HHVM* server at the *App. Server* tier and *Partial Post Replay* is used there to tackle any disruptions. Due to the very brief draining period for *HHVM* servers, *Socket Takeover* by itself is inadequate to prevent disruptions for large *POST* requests as these requests are not expected to be completed by the end of the draining phase for the old instance and hence would lead to connection resets when the old instance terminates. Therefore, a co-ordinated form of draining with the downstream is required to hand-over the request to another healthy instance. Additionally, operational aspects of *App. Server* tier makes *Socket Takeover* unfavorable as these machines are too constrained along CPU and memory dimensions to support two parallel instances (e.g., priming local cache for a new *HHVM* instance is memory-heavy). *Downstream Connection Reuse* is used at both *Edge* and *Origin Proxygen* for *MQTT*-based services due to their long-lived connection nature. For *Downstream Connection Reuse* to work at *Edge*, application code at the user-end needs to understand the connection-reuse workflow transactions.

The three mechanisms differ with respect to the protocol or the target layer in the networking stack. Hence, there's no interdependencies and the mechanisms are used concurrently. *Socket Takeover* and *Downstream Connection Reuse* are triggered at the same time, i.e., when a *Proxygen* restarts. Note that, if the next-selected machine to relay the MQTT connections is also under-going a restart, it does not have any impact on *Downstream Connection Reuse*, since the updated, parallel instance is responsible for handling all the new connections. For *Partial Post Replay*, it is possible that the next *HHVM* server is also restarting and cannot handle the partially-posted requested, since the corresponding instance is in draining mode and not accepting any new requests. In such a case, the downstream *Proxygen* retries the request with a different *HHVM* server. At production, the number of retries is set to 10 and is found enough to never result in a failure due to unavailability of active *HHVM* server.

5 HANDS-ON EXPERIENCE

In this section we discuss our experiences and the challenges we faced in developing and deploying the *Zero Downtime Release*.

5.1 Socket Takeover

As discussed in section 4.1, passing controls of a listening socket between processes with *SCM_RIGHTS* is known technique. Our solutions use it as a building block for zero-downtime release in a large-scale production service. It is thus important to understand limitations and operational downsides of using such feature and have proper monitoring and mitigation steps in place.

- **Downsides of sharing existing sockets:** When the ownership of a socket is transferred to another process its associated state within the kernel remains unchanged since the *File Descriptor (FD)* still points to the same underlying data structure for the socket within Kernel even though the application states in user-space have changed with the new process taking over the ownership of the socket.

While this mechanism to pass control of a socket is not an issue *per se*, an unchanged socket state in the Kernel even after restart of the associated application process is not only unintuitive but can also hinder in debugging of potential issues and prevent their swift mitigations. It is a common practice to roll back the newly released software to a last known version to mitigate ongoing issues. Albeit rare in occurrence, if there is any issue in the Kernel involving socket states, such as ones stemming from erroneous states within their data structure, diagnosing or mitigating such issues on a large scale fleet poses several challenges since a rollback of the latest deployment does not resolve the issue. For example, we encountered a bug in the UDP write path [22] after enabling the generic segmentation offload (UDP GSO) Kernel feature [23]. On many of the updated servers, the buffer in UDP sockets (*sk_buff*) were not cleared properly after failures within certain *syscalls* in write paths. This resulted in slow accumulation of buffers over period of time, eventually leading to a system wide failure to successfully write any further packet. A simple restart of the application process owning the socket did not mitigate this issue since the underlying socket structure persisted across process restarts while using the *Socket Takeover* mechanism.

Another pitfall with the mechanism of passing the control of sockets is that it introduces possibilities of leaking sockets and their associated resources. Passing the ownership of sockets to a different process using their *FDs* is essentially equivalent to a system call *dup(fd)* wherein upon passing these *FDs* to the new process, the Kernel internally increases their reference counts and keeps the underlying sockets alive even after the termination of the application process that owns them. It is thus essential that the receiving process acts upon each of the received *FDs*, either by listening on those sockets or by closing any unused ones. For example, when multiple sockets bind to the same address using the socket option *SO_REUSEPORT*, the Linux Kernel internally multiplexes incoming packets to each of these sockets in a fairly uniform manner. However, during a release if the newly spun process erroneously ignores any of the received *FDs* for these sockets after *Socket Takeover*, without either closing the *FDs* or listening on the received sockets for incoming packets, it can lead to large increase in user facing errors such as connection timeouts. This is because the orphaned sockets now owned by the new process are still kept alive in the Kernel layer and hence receive their share of incoming packets and new connections - which only sit idle on their queues and never get processed.

Remediation: For swift mitigations of operational issues during a release, such as the ones involving persistent erroneous socket

states or socket leaks as mentioned earlier in this section, a mechanism to dynamically disable the *Socket Takeover* process presents a safe and convenient approach. To allow such options, applications employing *Socket Takeover* mechanism must incorporate such fallback alternatives on all of its phases, spanning from the initialization phase (when sockets are created) to the shutdown phase (as the process gracefully terminates all of its existing connections). *Proxygen*, for example, allows transitioning to either modes by executing a set of heuristics tuned to its needs on each of its operating phases. During the initialization phase *Proxygen* resorts to allocating and binding new sockets if it cannot receive *FDs* of listening sockets from the existing process. Similarly, during its shutdown phase *Proxygen* adopts different strategies depending on the runtime behavior - such as signaling the *L4LB* to not send any new traffic by failing the health probes if no other instance is running on the same host, to transitioning back to the running mode from the shutdown phase if the newly spun process crashes during its initialization phase. Additionally, being able to selectively disable this approach for a subset of sockets makes ad-hoc debugging and development easier.

We recommend fine-grained monitoring of socket states, such as connection queue lengths, failure counts in socket reads/writes, number of packets processed and dropped and counts of *FDs* passed across process domains to detect problems of these nature in early phases. While the occurrence of such issues is quite rare in our experience, monitoring of such events not only notifies the concerned engineers but can also trigger automated responses to mitigate the impact.

• **Premature termination of existing process:** During a release phase, there is a period of time during which multiple application processes within a same host are serving requests. During such window of time, premature exit of the old instance before the new service instance is completely ready and healthy can result in a severe outage. Typically, the newly spun instance goes through initialization and setup phase, such as loading of configuration, setting up of its server pools and spinning of threads, and may also invoke remote service calls. It is thus common to encounter issues due to issues of its own such as deadlocks and memory leaks or due to issues external to itself, such as failures on the remote service calls in its startup path. It is thus imperative to keep the old service instance and serve incoming requests as the new instance is being initialized.

Remediation: Our recommendation based on experience is to implement a message exchange mechanism between the new and old application server instances to explicitly confirm that the *Socket Takeover* was successful, and acknowledge the readiness to serve. The same mechanism that was used to pass *FDs* of sockets during the *Socket Takeover* can be trivially extended to allow exchange of such messages. Only upon receiving the explicit message should the old instance initiate its shutdown phase with proper draining phase to allow graceful termination of existing connections.

• **Backward compatibility:** Any changes in the *Socket Takeover* mechanism must be backward compatible and support versioning. If the system of acknowledgment from the new instance to the old instance breaks, the service gets exposed to significant availability risk. To illustrate this notion further, any change in the behavior of the *Socket Takeover* version n needs to be compatible with the version $n+1$, and thus, needs to handle all the possible transitions:

– $n \rightarrow n$ restart of an existing version in production
– $n \rightarrow n+1$ deployment of the new implementation

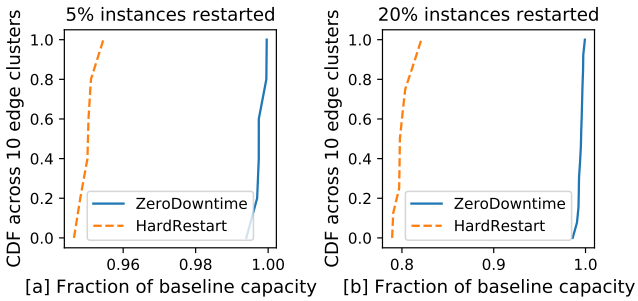


Figure (8) Cluster capacity at {5,20}% batch restarts.

– $n+1 \rightarrow n$ revert of the deployment above (for example, due to a regression)
– $n+1 \rightarrow n+1$ restart of the new version in production

For this reason we recommend the implementation of explicit versioning logic within the messaging mechanism to make it easier for the developer to specify the intended behavior when introducing new changes in the algorithm. A newly spun process can gracefully terminate itself early if it fails to negotiate a common version with an existing process. Only after successfully completing version negotiation, the existing process sends the *FDs* of listening sockets to the new process.

• **Availability risks and health-checks:** The process of release itself can increase the availability risks for a service going through the release. For instance, degradation in the health of a service being released even at a micro level, such as one at an individual host level, can escalate to a system wide availability risks while updating a large fraction of its servers. The overall release process, therefore, must be tuned such that the health of the service being updated remains consistent for an external observer of the service. For example, when *Proxygen*, an *L7LB* instance, is being updated with the *Socket Takeover* mechanism, its state must continue to be perceived as *healthy* to the observers in *L4LB* layer. Occasionally it is possible that the servers going through deployment in peak hours suffer momentary CPU and memory pressure, and consequently reply back as unhealthy to external health monitors for the service. This seemingly momentary flap can escalate to system wide instability due to mis-routing of packets for existing connections if, for example, the *L4LB* layer employs a consistent routing mechanism such as consistent-hash to pick an *L7LB* destination server based on the source and destination addresses in a packet header.

Remediation: To avoid instability in routing due to momentary shuffle in the routing topology, such as changes in the list of healthy servers going through a release process using the *Socket Takeover* mechanism, we recommend adopting a connection table cache for the most recent flows. In Facebook we employ a *Least Recently Used (LRU)* cache in the *Katran* (*L4LB* layer) to absorb such momentary shuffles and facilitate connections to be routed consistently to the same end server. Adoption of such mechanism also usually yields performance improvements.

5.2 Partial Post Replay

Here we discuss potential pitfalls of the store-and-replay solution regarding HTTP semantics and app. server behavior; while some of the solutions were part of the original design, some others are less

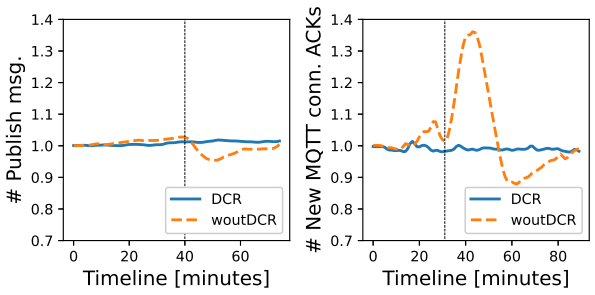


Figure (9) Impact of *Downstream Connection Reuse*

obvious and were only discovered after deploying the solution in production.

• **Preserving HTTP Semantics:** Partial Post Replay is designed to work with any HTTP version; some simple rules must be defined for each protocol version to make sure that the necessary state is transferred back to the proxy so that the original request can be replayed to a different server. As an example HTTP/2 and HTTP/3 request pseudo-headers (beginning with $':'$) are echoed in the response message with a special prefix (e.g. $'pseudo-echo-path:'$ for the $':path:'$ pseudo-header). The most interesting corner cases however were discovered with HTTP/1.1 and chunked transfer encoding where the body stream is split into chunks; each chunk is preceded by a header that indicates the chunk length and the end of the body is signaled by a chunk trailer. A proxy implementing PPR must remember the exact state of forwarding the body to the original server, whether it is in the middle or at the beginning of a chunk in order to reconstitute the original chunk headers or recompute them from the current state.

• **Trust the app. server, but always double-check:** A solution like PPR requires a Proxy and its immediate upstream hop to collaborate and implement the client and server-side of the mechanism. In Facebook infrastructure since we control both sides there is implicit trust on the app. server doing the right thing and not be malicious. However, the upstream may also behave as a proxy itself forwarding responses from another app. server which does not implement PPR and may be using the HTTP response status code 379. We hit this case in production, where the web-server acting as a proxy would return responses from a buggy upstream service returning randomized HTTP response codes due to a memory corruption bug. Although this was due to a bug, we realized that there was the need for a more strict check on the conditions to enable the feature on a specific request.

Remediation: The current implementation and RFC do not define a negotiation mechanism for the feature and assumes previous knowledge at the intermediary and server that the peer is going to support the feature. Also, HTTP response code 379 was specifically picked within an unreserved range in the IANA status code registry [6] and therefore no assumption can be made on the server not using that status code for other purposes. To disambiguate then we used the HTTP Status message, and defined that the proxy must enable PPR only on seeing a 379 response code *with PartialPOST* as the status message.

6 EVALUATION

Our evaluation of our framework, *Zero Downtime Release*, is motivated by the following practical questions:

(1) How does *Zero Downtime Release* fare in comparison with traditional release techniques on performance and availability grounds?
(2) What are the operational benefits of using *Zero Downtime Release* at productions scale in terms of minimizing disruptions, preserving capacity and release scheduling?
(3) What are the system overheads of *Zero Downtime Release*?

Evaluation Metrics *Zero Downtime Release* has been operational at Facebook for multiple years and has assisted in rolling-out thousands of code updates with minimal disruptions. A sophisticated auditing infrastructure [37, 49] has been built over the years for real-time monitoring of cluster and user performance, including releases and their implications. Each restarting instance emits a signal through which its status can be observed in real-time (e.g., health of the parallel processes, duration of takeover etc.). The instances also log system benchmarks (e.g., CPU utilization, throughput, Request per Second (RPS) served etc.) as well as counters for the different connections (e.g., Number of MQTT connections, HTTP status code sent, TCP RSTs sent etc.). The monitoring systems also collect performance metrics from the end-user applications and serve as the source of measuring client-side disruptions (e.g., errors, HTTP codes sent to user etc.). Our evaluation of *Zero Downtime Release*, we examine these data sources to analyze the performance and operational implications of our *Zero Downtime Release* framework.

Evaluation Setup In our evaluation, we conduct experiments in production clusters across the globe, serving live end-user traffic. Experimenting with live deployments allows us to not only measure the impact at scale, but also measure the impacts across the different protocols. For each system component, we aim to highlight improvement in target system's availability, quality of service (QoS) and their impact on client. We further explore their use in alleviating the complexity of managing hundreds of production clusters. Finally, we address the additional costs related to persistent long haul techniques and explore their impact on performance.

6.1 Comparison with Traditional Release

To measure the effectiveness of *Zero Downtime Release*, we conducted multiple *HardRestart* across 10 productions clusters. A *HardRestart* mirrors the traditional roll-out process — updates are rolled out in batches across a cluster and the restarting instances enter the draining mode (i.e., the server stops receiving new connection until the end of draining period). Since the goal is to compare against *Zero Downtime Release*, we set the same draining duration (20 minutes) and test two batch sizes (5% and 20%) in ten randomly selected *Edge* production clusters. During both restart strategies, we monitor system metrics (e.g., idle CPU) and performance counters (e.g., HTTP and MQTT stats). Furthermore, we analyzed the client-side disruptions by examining performance metrics collected from end-users.

6.1.1 ***Improved time to completion.*** Figure 16 summarizes *Completion Times* of various restart mechanisms for *Proxygen* and *App. Server* releases (i.e., time required to update our global deployments for either *Proxygen* and *App. Server*). We observe that in the median update, *Proxygen* releases finish in 1.5 hours, whereas, *App. Server* releases are even faster (25 minutes). The major factor behind the differences in their completion time is the different draining behavior. *Proxygen* are configured to drain for 20 minutes while *App. Server* have a short draining interval (10-15 seconds) since their workload is dominated by short-lived requests. As we are going to

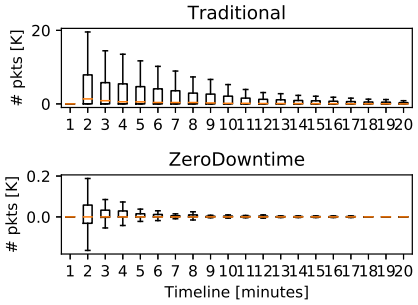


Figure (10) Packet mis-routing

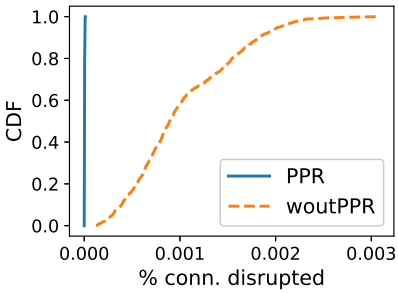


Figure (11) POST disruption

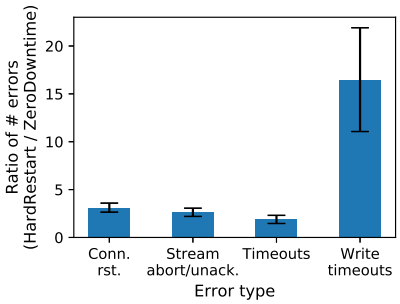


Figure (12) Proxy errors comparison

show next, *Zero Downtime Release* preserves capacity and minimizes while taking order of tens of minutes to restart the tiers.

6.1.2 ***Improved L7 Cluster Capacity.*** *Katran* maintains an updated view of available *Proxygen* through health-checks. Recall that performing a HardRestart on an instance causes this instance to block new connections and thus to fail health-checks, because health-check connections are rejected. Whereas *Zero Downtime Release* enables the new *Proxygen* instance to take-over health-check responsibility. Looking at *Katran* logs, we observe the expected behavior: *Zero Downtime Restart* stays transparent to *Katran* while, for HardRestart, the restarted instances are removed from *Katran* table.

To explore the impact of the two restart approaches on clusters' available capacity, we measure the idle CPU metrics under the draining phase of both restart approaches. Figure 8(b) plots the cluster's total idle CPU resources, normalized by the baseline idle CPU resources, recorded right before the restart. In *Socket Takeover* (§ 6.3), we expect an increase in CPU usage because of the parallel process on same machine, leading to a slight (within 1%) decrease in cluster's idle CPU. However, this is radically different from the HardRestart case, where the cluster's CPU power degrades linearly with the proportion of instances restarted because each instance is completely taken offline.

6.1.3 ***Minimizing User Faced Disruptions.*** **Pub/Sub services (*Downstream Connection Reuse*):**

To measure MQTT related disruptions, we performed restarts with and without *Downstream Connection Reuse* (DCR at the *Origin*. Figure 9 highlights its impact on minimizing the client side disruptions. The figure plots a timeline of Publish messages routed through the tunnel to measure the impact of restarts on communication between end-users and their MQTT brokers (back-ends). The figure also plots the median number of new MQTT connections created at the back-ends, by measuring the number of ACKs sent in response to MQTT connect messages from end-users. The number represent the median across the cluster machines and are normalized by their value right before restart. In contrast to DCR case where the number of published messages do not deteriorate during the restart, we observer a sharp drop in Publish messages when *Downstream Connection Reuse* is not used (woutDCR), indicating disruptions in communication between users and their MQTT brokers. On the other hand, we observe a sharp spike in number of ACKs sent for new MQTT connections for woutDCR case, indicating that the restarting instance terminated the MQTT connection, leading to the clients retrying to reconnect with the back-end brokers. With DCR, we do not observe any change as connections between users and their back-end brokers are not terminated and, instead, are routed through another *Origin Proxygen* to same broker.

Web (*Partial Post Replay*): In absence of *Partial Post Replay* (PPR), the restarting *App. Server* terminates the in-process POST requests and returns error code (e.g., HTTP code 500) to the downstream *Proxygen* and, eventually, the error code reaches the end-user. In case of PPR, the server returns a code 379 and the partial POST data which is then replayed to another *App. Server* alongside the original request.

To test *Partial Post Replay*'s effectiveness, we observe *App. Server* restarts from the downstream *Origin Proxygen*'s vantage point and inspect the POST requests sent to a restarting server. A reception of 379 response, along with the partial request data, signals a request that would have faced disruption in the absence of *Partial Post Replay*—allowing us to measure the scale of disruptions due to *App. Server* restarts. Figure 11 compares the *Partial Post Replay*'s impact by presenting the percentage of connections disrupted across the web tier for 7 days. Note that *App. Server* are restarted tens of times a day (Figure 2a) and the 7 days worth of data covers around 70 web tier restarts. We observe that *Partial Post Replay* is extremely effective at minimizing the POST requests disruptions. Although the percentage might seem very small (e.g., 0.0008 at median), there are billions of POST request per minute for the web-tier and even the small percentages translate to huge number of requests (e.g, ~6.8 million for median).

6.1.4 ***Minimizing Proxy Errors.*** A major benefit of using *Zero Downtime Release* is to improve proxy performance during restart w.r.t. errors. Errors result in connection terminations or 500 response codes both of which are highly disruptive to end-user's performance and QoE. To measure these disruptions, we measured the errors sent by the *Edge* proxy to end-users, under both kind of restarts. Figure 12 presents the ratio of errors observed for the two restarts (traditional and *Zero Downtime Release*). The 4 types of errors correspond to different types of disruptions: (i) Connection Reset (conn. rst.) refers to sending a TCP RST to terminate the connection, (ii) Stream abort/unacknowledged refers to errors in HTTP, (iii) Timeouts refer to TCP level timeouts, (iv) write timeout refers to case when application times-out due to disruption in underlying connection. We observe a significant increase in all errors for "traditional" as compared to *Zero Downtime Release*. Write timeouts increase by as much as 16X and are significantly disruptive for user experience as users can not retry right away.

6.1.5 ***Impact on consistent packet routing.*** Next, we measure the efficacy of *Socket Takeover* for consistently routing packets to the right proxy process, in cases where multiple proxies are available (updated and draining instance). We disable the connection-ID based

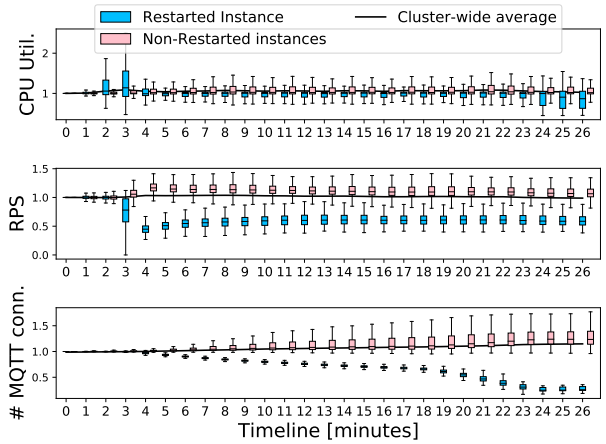


Figure (13) Timeline of *Proxygen* restart

QUIC packet routing and inspect the packets received at both instances. Since the HardRestart case has only one instance running at a time, no UDP mis-routing exists. In the context of this experiment, *traditional* approach refers to the case where sockets are migrated to the updated instance, but the system lacks connection-ID powered user-space routing. Figure 10 present the number of UDP packets mis-routed per instance. A packet is marked mis-routed if the wrong proxy instance receives it i-e packets bound for the draining instance are received at updated one. Although we observe some mis-routing for *Zero Downtime Release* at start of the restart, their magnitude is insignificant compared to traditional case, with 100X less packets mis-routed for the worst case (tail at T=2).

6.2 Operational Benefits at Scale

To evaluate the effectiveness of *Zero Downtime Release* at scale, we monitored 66 production cluster restarts across the globe.

6.2.1 ***Performance and stability improvements:*** Figure 13 shows a timeline of the system and performance metrics (Requests Per Second (RPS), number of active MQTT conn., throughput and CPU utilization) during releases. The metrics are normalized by the value just before the release was rolled-out. During each batch restart (20% of the instances), we collected the target metrics from each cluster instance and Figure 13 plots the distributions (averaged over a minute) observed across two groups of machines: (i) the 20% restarted (G_R), (ii) the rest of 80% non-restarted (G_{NR}). Observing the two groups side by side demonstrates standing their behavior during restarts. The timeline (x-axis in minutes) marks 4 phases: (i) T≤1 state before restart, (ii) T=2 marks the start of batch restart, (iii) T=24 marks the end of draining period, (iv) T≥24 state after batch restart is concluded. All the observed metrics are normalized by their values at T=0. We further present a cluster-wide view in form of the average metrics calculated across all instances of the cluster.).

Cluster-wide behavior: Across RPS and number of MQTT conn., we observe virtually no change in cluster-wide average over the restart period. No significant change in these cluster-wide metrics after T=2, even with 20% of the cluster restarting, this highlights the benefits of *Zero Downtime Release* at scale in practice. We do observe a small increase in CPU utilization after T=2, attributed to the system overheads of *Socket Takeover*, i.e., two *Proxygen* instances run parallel for the duration of draining period on same machine resources (§ 6.3).

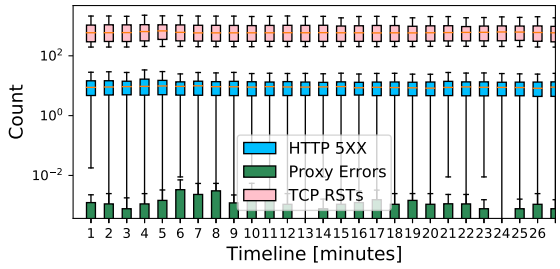


Figure (14) Disruptions during *Proxygen* restart

G_R vs G_{NR} behavior: Analyzing the per-group breakdown, we observe the inflation in CPU utilization for restarting instances (G_R) only persists for two minutes (at T=2,3) and the CPU util. gradually decreases until the end of draining period where we observe a sharp decrease (at T≥24) due to termination of parallel process. CPU util. of the (G_R) to be lower than cluster-wide average and G_R (non-restarted group) is surprising as every machine in G_R runs two *Proxygen* instances during $2 \leq T \leq 24$. We observe RPS to drop for G_R and rise for G_{NR} after T=3, indicating that G_R instances are serving lower number of requests than their pre-restart state and the G_{NR} instances are serving a higher proportion. The contrasting behavior for the two groups arise due to CPU being a function of RPS i-e an instance serving less number of requests requires lower CPU cycles. Since *Katran* uniformly load-balances across *Proxygen* fleet and the newly-spun, updated instance has no request backlog, it gets the same share of requests as others — leading to the drop and ramp-up in RPS over time.

For MQTT connections, we observe their number to fall across G_R instances and gradually rise for G_{NR}. This behavior is expected as the MQTT connections get migrated to other healthy instances (G_{NR}) through DCR. However, we do not observe their number to drop to zero for G_R at end of draining as the updated, parallel *Proxygen* picks up new MQTT connections during this duration.

Timeline for disruption metrics: Figure 14 builds a similar timeline for disruption metrics – TCP resets (RST), HTTP errors (500 codes) and Proxy errors, presenting their count. Each point is the average value observed for a cluster and the box plot plots the distribution across the clusters. The timeline is divided into the four phases, similar to Figure 13. We observe that the disruption metrics stay consistent throughout the restart duration. Even for a 20% restart, we do not observe any increase in these disruption metrics — highlighting the effectiveness of *Zero Downtime Release* in shielding disruptions. No change in TCP RSTs highlights the efficacy of *Zero Downtime Release* for preventing TCP SYN related inconsistencies, observed for *SO_REUSEPORT* based socket takeover techniques [38].

6.2.2 ***Ability to release at peak-hours.*** Traffic load at a cluster changes throughout the day (exhibiting di-urinal pattern [44]). The traditional way is to release updates during off-peak hours so that the load and possible disruptions are low. Figure 15 plots the PDF of *Proxygen* and *App. Server* restarts over the 24 hours of the day. *Proxygen* updates are mostly released during peak-hours (12pm-5pm). Whereas, the higher frequency of updates for *App. Server* (Figure 2a) leads to a continuous cycle of updates for the *App. Server* — a fraction of *App. Server* are always restarting throughout the day as seen by the flat PDF in Figure 15. From an operational perspective, operators are expected to be hands-on during the peak-hours and the ability to release during these hours go a long way as developers can swiftly investigate and solve any problems due to a faulty release.

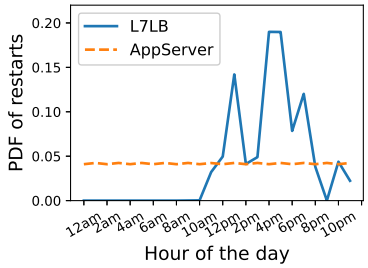


Figure (15) Update release time

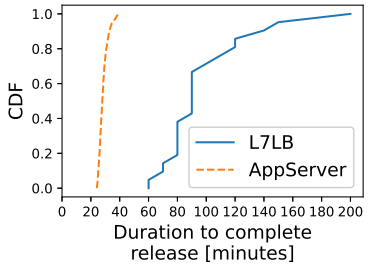


Figure (16) Time required to push release

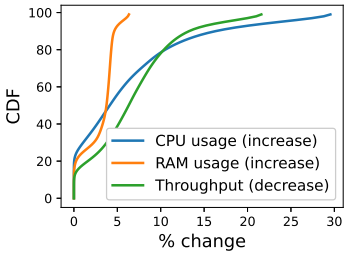


Figure (17) System benchmarks

6.3 System Overheads

Micro-benchmarks: Improving cluster availability and client's performance during a restart can come at the cost of increased system resource usage. Figure 17 plots the system resource usage during the restart phase for machines in a randomly chosen production edge cluster. Since the CPU consumption is variable at different phases of the restart (increases at first and then returns to normal state as seen in timeline figure 13), we plot system benchmarks during the entire restart and present the median numbers observed across the different machines in a randomly selected edge cluster. The presence of two concurrent *Proxygen* instances contributes to the costs in system resources (increased CPU and Memory usage, decreased throughput). The change in throughput correlates with CPU usage (inverse proportionally), and the tail throughput decreases is caused by the initial spike in CPU usage. Although the tail resource usage can be high (persisting for around 60-70 seconds), the median is below 5% for CPU and RAM usage i-e the increased resource usage does not persistent for the whole draining duration (§ 6.2). As the machine is still available and able to serve connections, this overhead is a small price to pay for minimizing disruptions and keeping the overall cluster close to its baseline capacity (i.e., non-restart scenario).

7 RELATED WORK

Release engineering: [24] is critical to important and performance for large-scale systems and has been a topic of discussion among industry [8, 20, 21], and researchers as well [9]. Different companies handle the release process in their own way, to suit the needs of their services and products [9, 20]. With the increased focus on improving performance and availability of global-scale systems, release engineering has become a first class citizen [20] at present and this paper moves forward the discussion around the best practices for update releases at a global scale.

Load-balancing at scale: The paper builds on the recent advancements in improving network infrastructure (Espresso [58], FBOSS [19], Katran [7] etc) and the use of L7LB [2, 3, 48, 53, 56] for managing traffic, to proposes novel ideas to improve the stability and performance by leveraging the L7LB, in order to mask any possible restart-related disruptions from end-users and improve the robustness of protocols that do not natively support graceful shutdowns. Note that, some aspects of the papers have been discussed in earlier works [34, 46] from Facebook. However, this paper tackles a wider-range of mechanisms, evaluate them at production-scale and describes the experiences of using these mechanisms.

Managing restarts and maintenance at scale: Recent work has focused on managing failures and management-related restart/updates for various components of infrastructure, ranging from hardware repairs [57] to network and switch upgrades [10, 32] in data-centers. Our focus is mostly on software failures and graceful handling of restarts, to allow faster deployment cycles and have zero disruptions. While these works focus on mostly data-center scenarios, we tackle the entire end-to-end infrastructure at a global scale.

Disruption avoidance: Recently, a few proxies have been armed with disruption avoidance tools to mitigate connection terminations during restarts [36, 43]. HAProxy proposed *Seamless Reloads* [35, 55] in 2017 and socket FD transfer mechanism (similar to *Socket Takeover*) was added in 2nd half 2018 [4]. Envoy recently added *Hot Restart* [16] that uses a similar motivation. Our mechanisms are more holistic as they support disruption-free restarts for protocols other than TCP (e.g., UDP) and provide an end-to-end support to mitigate disruptions, e.g., *Partial Post Replay* for HTTP and *Downstream Connection Reuse* for MQTT. Additionally, this work provides a first-time, hands-on view of the deployment of these techniques on a global scale.

8 CONCLUSION

Owing to high code volatility, CSPs release upto tens of updates daily to their millions of globally-distributed servers and the frequent restarts can degrade cluster capacity and are disruptive to user experience. Leveraging the end-to-end control over a CSP's infrastructure, the paper introduces *Zero Downtime Release*, a framework to enable capacity preservation and disruption-free releases, by signaling and orchestrating connection hand-over during restart (to a parallel process or upstream component). The framework enhances pre-existing kernel-based mechanisms to fit diverse protocols and introduces novel enhancements on implementation and protocol fronts to allow fast, zero-downtime update cycles (globally-distributed fleet restarted in 25 minutes), while shielding millions of end-users from disruptions.

9 ACKNOWLEDGMENTS

Many people in the Proxygen and Protocols teams at Facebook have contributed to *Zero Downtime Release* over the years. In particular, we would like to acknowledge Subodh Iyengar and Woo Xie for their significant contributions to the success of *Zero Downtime Release*. We also thank the anonymous reviewers for their invaluable comments. This work is supported by NSF grant CNS-1814285.

REFERENCES

[1] Django. https://www.djangoproject.com/.
[2] Envoy Proxy. https://www.envoyproxy.io/.
[3] HAProxy. http://www.haproxy.org/.
[4] HAProxy source code. https://github.com/haproxy/haproxy.
[5] HHVM. https://github.com/facebook/hhvm.
[6] Hypertext Transfer Protocol (HTTP) Status Code Registry. https://bit.ly/3gqRrtP.
[7] Katran - A high performance layer 4 load balancer. https://bit.ly/38ktXD7.
[8] Bram Adams, Stephany Bellomo, Christian Bird, Tamara Marshall-Keim, Foutse Khomh, and Kim Moir. 2015. The practice and future of release engineering: A roundtable with three release engineers. *IEEE Software* 32, 2 (2015), 42–49.
[9] Bram Adams and Shane McIntosh. 2016. Modern release engineering in a nutshell–why researchers should care. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 5. IEEE, 78–90.
[10] Omid Alipourfard, Chris Harshaw, Amin Vahdat, and Minlan Yu. 2019. Risk based Planning of Network Changes in Evolving Data Centers. (2019).
[11] Sid Anand. 2011. Keeping Movies Running Amid Thunderstorms Fault-tolerant Systems @ Netflix. QCon SF. https://bit.ly/37ahP65
[12] Oracle Corporation and/or its affiliates. Priming Caches. https://bit.ly/2OxnPzi.
[13] AppSignal. 2018. Hot Code Reloading in Elixir. https://bit.ly/2H1k8hh.
[14] Envoy Project Authors. Command line options, drain-time-s. https://bit.ly/38f3RRW.
[15] Envoy Project Authors. Command line options, parent-shutdown-time-s. https://bit.ly/2Sa7Fy5.
[16] Envoy Project Authors. Hot restart. https://bit.ly/2H3Kwan.
[17] The Kubernetes Authors. Safely Drain a Node while Respecting the PodDisruptionBudget. https://bit.ly/2SpYgkZ.
[18] Netflix Technology Blog. 2018. Performance Under Load. https://bit.ly/2OCQbYU.
[19] Sean Choi, Boris Burkov, Alex Eckert, Tian Fang, Saman Kazemkhani, Rob Sherwood, Ying Zhang, and Hongyi Zeng. 2018. Fboss: building switch software at scale. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 342–356.
[20] SE Daily. Facebook Release Engineering with Chuck Rossi - Transcript. https://bit.ly/2H8Xwew.
[21] SE Daily. 2019. Facebook Release Engineering with Chuck Rossi. https://bit.ly/3bfGaL7.
[22] Willem de Bruijn. udp: with udp segment release on error path. http://patchwork.ozlabs.org/patch/1025322/.
[23] Willem de Bruijn and Eric Dumazet. 2018. Optimizing UDP for content delivery: GSO, pacing and zerocopy. In *Linux Plumbers Conference*.
[24] Andrej Dyck, Ralf Penners, and Horst Lichter. 2015. Towards definitions for release engineering and DevOps. In *2015 IEEE/ACM 3rd International Workshop on Release Engineering*. IEEE, 3–3.
[25] Alex Eagle. 2017. You too can love the MonoRepo. https://bit.ly/2H40EbF.
[26] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. 2016. Maglev: A fast and reliable software network load balancer. In *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*. 523–535.
[27] Alan Frindell. HTTP Partial POST Replay. https://tools.ietf.org/html/draft-frindell-httpbis-partial-post-replay-00.
[28] GlobalScape. Server Drain, Maintenance, and Auto-Restart. https://bit.ly/31B7A9A.
[29] Sara Golemon. 2012. Go Faster. https://bit.ly/2Hg1Ed7.
[30] Christian Hopps et al. 2000. *Analysis of an equal-cost multi-path algorithm*. Technical Report. RFC 2992, November.
[31] Jez Humble. 2018. Continuous Delivery Sounds Great, but Will It Work Here? *Commun. ACM* 61, 4 (March 2018), 34–39.
[32] Michael Isard. 2007. Autopilot: automatic data center management. *ACM SIGOPS Operating Systems Review* 41, 2 (2007), 60–67.
[33] Jana Iyengar and Martin Thomson. 2018. Quic: A udp-based multiplexed and secure transport. *Internet Engineering Task Force, Internet-Draft draftietf-quic-transport-17* (2018).
[34] Subodh Iyengar. 2018. Moving Fast at Scale: Experience Deploying IETF QUIC at Facebook. In *Proceedings of the Workshop on the Evolution, Performance, and Interoperability of QUIC* (Heraklion, Greece) *(EPIQâĂŹ18)*. Association for Computing Machinery, New York, NY, USA, Keynote.
[35] Lawrence Matthews Joseph Lynch. Taking Zero-Downtime Load Balancing even Further. https://bit.ly/2OA66XY.
[36] Michael Kerrisk. The SO_REUSEPORT socket option. https://lwn.net/Articles/542629/.
[37] Ran Leibman. Monitoring at Facebook - Ran Leibman, Facebook - DevOpsDays Tel Aviv 2015. https://bit.ly/2ODbgT1.
[38] Joseph Lynch. True Zero Downtime HAProxy Reloads. https://bit.ly/31H2dWz.
[39] Linux man page. cmsg(3) - access ancillary data. https://linux.die.net/man/3/cmsg.
[40] Linux man page. dup, dup2, dup3 - duplicate a file descriptor. https://linux.die.net/man/2/dup.
[41] Linux man page. recvmsg(2). https://linux.die.net/man/2/recvmsg.
[42] Linux man page. sendmsg(2). https://linux.die.net/man/2/sendmsg.
[43] Suresh Mathew. Zero Downtime, Instant Deployment and Rollback. https://bit.ly/2ZgNGzV.
[44] Arun Moorthy. 2015. Connecting the World: A look inside FacebookâĂŹs Networking Infrastructure. https://unc.live/2UzVe0f.
[45] mqtt.org. MQ Telemetry Transport, machine-to-machine (M2M) connectivity protocol. http://mqtt.org/.
[46] Kyle Nekritz and Subodh Iyengar. Building Zero protocol for fast, secure mobile connections. https://bit.ly/2VkkoiH.
[47] Sam Newman. 2015. *Building microservices: designing fine-grained systems*. " O'Reilly Media, Inc.".
[48] NGINX. NGINX Reverse Proxy. https://bit.ly/39fkWLt.
[49] Inc OâĂŹReilly Media. 2012. FacebookâĂŹs Large Scale Monitoring System Built on HBase. https://bit.ly/2tAHlnc.
[50] Chuck Rossi. 2017. Rapid release at massive scale. https://bit.ly/2w0T9jB.
[51] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 21–30.
[52] Amazon Web Services. Configure Connection Draining for Your Classic Load Balancer. https://amzn.to/39iQlMS.
[53] Daniel Sommermann and Alan Frindell. 2014. Introducing Proxygen, Facebook HTTP framework.
[54] Facebook Open Source. Hack - Programming Productivity Without Breaking Things. https://hacklang.org/.
[55] Willy Tarreau. 2017. Truly Seamless Reloads with HAProxy âĂŞ No More Hacks! https://bit.ly/31Ihfvm.
[56] VDMS. Our software - CDN. https://bit.ly/2UC0kZI.
[57] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. 2010. Characterizing cloud computing hardware reliability. In *Proceedings of the 1st ACM symposium on Cloud computing*. ACM, 193–204.
[58] Kok-Kiong Yap, Murtaza Motiwala, Jeremy Rahe, Steve Padgett, Matthew Holliman, Gary Baldus, Marcus Hines, Taeeun Kim, Ashok Narayanan, Ankur Jain, et al. 2017. Taking the edge off with espresso: Scale, reliability and programmability for global internet peering. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 432–445.