Bladerunner: Stream Processing at Scale for a Live View of Backend Data Mutations at the Edge

Jeff Barber,[†] Ximing Yu,[†] Laney Kuenzel Zamore,[†] Jerry Lin,[†] Vahid Jazayeri,[†]
Shie Erlich,[†] Tony Savor,[†*] and Michael Stumm[‡]
[†] Facebook [‡] University of Toronto

Abstract

Consider a social media platform with hundreds of millions of online users at any time, utilizing a social graph that has many billions of nodes and edges. The problem this paper addresses is how to provide each user a continuously fresh, up-to-date view of the parts of the social graph they are currently interested in, so as to provide a positive interactive user experience. The problem is challenging because the social graph mutates at a high rate, users change their focus of interest frequently, and some mutations are of interest to many online users.

We describe Bladerunner, a system we use at Facebook to deliver relevant social graph updates to user devices efficiently and quickly. The heart of Bladerunner is a set of backend stream processors that obtain streams of social graph updates and process them on a per application and per-user basis before pushing selected updates to user devices. Separate stream processors are used for each application to enable application-specific customization, complex filtering, aggregation and other message delivery operations on a per-user basis. This strategy minimizes device processing overhead and last-mile bandwidth usage, which are critical given that users are mostly on mobile devices.

CCS Concepts: • **Information systems** → **Data streaming**; *Stream management*; *Service buses*; *Social networks*; • **Computer systems organization** → **Cloud computing**; • **Networks** → **Session protocols**; • **Software and its engineering** → **Publish-subscribe / event-based architectures**; • **Applied computing** → *Event-driven architectures.*

Keywords: Internet-scale data dissemination, publish-subscribe, mobile cloud infrastructure, event-driven stream processing, request-stream protocol, scalability, social networks

[*] Savor is now employed by Google.

SOSP '21, October 26–29, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-8709-5/21/10.
https://doi.org/10.1145/3477132.3483572

1 Introduction

Facebook operates a social media platform with hundreds of millions of connected users from around the world at any instance of time. Our platform utilizes a social graph that has many billions of nodes and edges. Its data is distributed and replicated across a number of globally distributed data centers. Multiple layers of caches are used to provide low latency access to the data and to reduce the load on backend storage servers. Facebook's social graph storage and caching system is called ***TAO*** [13].

Given the scale we operate at and the fact that the social graph is continuously mutating, the fundamental problem we face is how to provide every one of our end-users with fresh and up-to-date data of targeted interest on their devices[1] so as to ensure a positive interactive user experience. In particular, new updates that add, remove, or modify data to areas of the social graph that are of current interest to any user must reach the user's devices in as real-time as possible. How best to do this is the focus of this paper.

Challenges. The problem of keeping data of interest fresh on devices is particularly challenging in our environment because of the following factors:

1. Social graph mutability and relevance. Parts of the social graph can mutate at a high rate — e.g., a live video broadcast with one hundred million live viewers can result in over one million live comments submitted per second, at peak. At the same time, new graph data can be time sensitive — e.g., comments to a live video become uninteresting to the user relative quickly.

2. Rapidly changing foci. Users change the focus of their interest frequently; e.g., a user scrolling through displayed posts in their News Feed is primarily interested in comments related to the post they are currently focused on.

3. Legacy substrate. Most of the clients are on mobile devices, and many of these devices have limited processing and storage capacity — more than 60% of the world's mobile phones have less processing and memory capacity than state-of-the-art phones of 6 years ago. Further, many parts of the world still operate with older mobile communication infrastructure where 50%+ of the users are limited to 2G infrastructure; thus, mobile data bandwidth is often limited, and many end-users must pay for the bandwidth they use. Even with modern devices in 5G environments, the quality

[1] We use the term *device* to refer both to mobile devices and browsers.

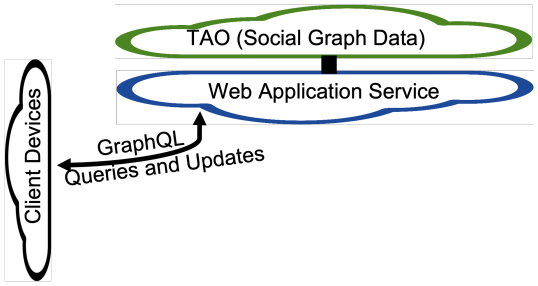

Figure 1. Polling for social graph updates.

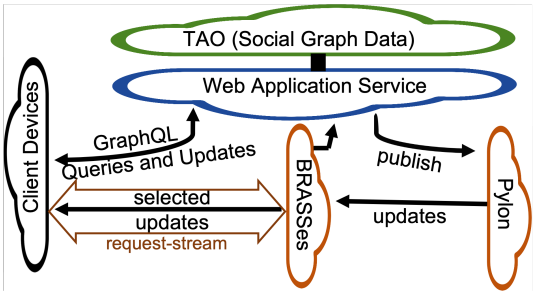

Figure 2. Bladerunner high-level structure and data flow.

of mobile connectivity can vary significantly depending on the location of the end-user; disconnections are common.

Polling solution. A straightforward approach to keep targeted data fresh, and one we used for a number of years, is to have devices periodically poll the backend infrastructure for updates to the social graph using a query language such as GraphQL [24] (see Fig. 1). This solution is attractive because it is easy to implement client-side, and its request-response model easily copes with server and connection failures.

However, we found polling to be problematic for three reasons. First, it wastes network bandwidth and device battery, because in our environment 80% of the queries return no new data, as no new updates have occurred since the previous poll. While bandwidth and battery usage can be managed to some degree by regulating the polling frequency, in practice it is difficult to decide on an effective frequency given that data updates often occur in unpredictable bursts, and delayed data is less engaging to the user.

Second, frequent polling (i.e., every 1-2 seconds) burdens the backend infrastructure with superflous overhead — in our case on the order of a billion queries per second that return no new data but incur the overhead of the queries. Making matters worse, these queries are typically range and intersect[2] queries which incur particularly high overheads on the backend and typically require querying multiple storage/caching servers. When we first rolled out support for *LiveVideoComments* with an implementation based on polling, our infrastructure team struggled to expand physical infrastructure rapidly enough to support the load from the *LiveVideoComments* service as it scaled in use, in part because the system has to be provisioned for peak load.

Thirdly, polling introduces extra delays in getting the relevant data to the device, with the delay being a function of the polling frequency, the load on the backend servers, and the communication latency between device and server. These extra delays negatively impact the end-user experience; e.g., comments on an event in the live video delivered late are far less interesting and relevant to the user than timely ones.

A better solution was needed, one that (*i*) pushes relevant social graph updates to end-user devices as they occur, in order to significantly reduce the amount of polling; (*ii*) shifts as much of the client processing of the data updates from the devices to the backend; and (*iii*) pushes data to devices only if it is likely to be displayed to the end-user in order to minimize last-mile communication bandwidth. Over the years, we had either used or tried to use various existing approaches to data dissemination, but, as we describe in §2, we found that they all had serious drawbacks for our applications when used at scale. This caused us to go back to the drawing board and design a system that met the requirements above.

Bladerunner is the system we designed, implemented, and now use to deliver relevant social graph updates to end-user devices *efficiently* and in *real-time* (i.e., within a few seconds). Over 100 applications[3] have been onboarded to use Bladerunner since it was first deployed. Some of the more prominent ones, besides *LiveVideoComments*, include: *LiveVideoReactions*, *Messenger* content delivery, *NewsFeedPostComments* and *NewsFeedPostLikes*, *WebsiteNotifications*, per user *StoriesTray* and *RoomTray* updates, *ActiveStatus* identifying friends that are currently online, and *TypingIndicator* (e.g., in *NewsFeedPostComments* and *Messenger*).

At a high level, the Bladerunner architecture is comprised of (*i*) a set of backend streaming servers called ***BRASSes*** that decide, on a per-client and per-application basis, what data to push to client devices and at what rate; (*ii*) a simple and highly scalable backend publish/subscribe system called ***Pylon*** to disseminate social graph updates to the BRASS streaming servers; and (*iii*) a request-stream protocol that connects client devices to the BRASS streaming servers.

As shown in Fig. 2, each new update to the social graph (e.g., a new *LiveVideo* comment) is created by an app running on a user's device and then sent from there to a target Web Application Server (**WAS**), which in turn issues the update request to TAO for storage. (This is the same WAS that poll requests are sent to.) Bladerunner, however, additionally requires the WAS to also "publish" (i.e., push) the update to Bladerunner's topic-based pub/sub service, Pylon, using a topic that identifies the part of the social graph that was updated.[4] Pylon then pushes the update to the streaming

[2] Intersect queries (e.g., SQL's INTERSECT operator) return the intersection of two or more queries.

[3] We use the term "application" to refer to a Bladerunner use case. An app running on a device may use Bladerunner in multiple different ways, each referred to as an application.

[4] This is similar in spirit to in-memory caching solutions that require applications to mirror their DB writes to the cache.

servers that expressed interest in receiving updates published to that topic by having previously issued a subscribe request.

BRASSes (Bladerunner Application Stream Servers) are responsible for obtaining relevant tsocial graph updates from Pylon; filtering them on a per-user basis based on relevance, timeliness, and privacy; and then pushing selected updates to the user devices. Thus, Pylon and BRASSes together implement two levels of fanout for each update, which is necessary given the scale we operate at. A key benefit of BRASSes is that they offload client-specific processing from end-user devices, and they reduce last-mile communication bandwidth usage by pushing only data to the device that will likely be made visible to the end-user.

Devices issue a *subscribe* request to an appropriate BRASS for each area of the social graph they wish to receive updates from. Each such request instantiates a *request-stream*, a core Bladerunner abstraction for an application-level connection between device and BRASS over which updates are delivered. A separate stream is instantiated for each part of the social graph the device wishes to obtain updates from. For each stream, the device specifies its interest using a framework similar to GraphQL Subscriptions [48] or GraphQL Live Queries [44].

A major benefit of this path for updates — from device to WAS to Pylon to BRASSes and from there to interested devices — is that it substantially reduces TAO query overhead because it significantly reduces the need for polling. When the *LiveVideoComments* service was switched over in production from its polling implementation to its Bladerunner implementation, the measured CPU load on the Web application servers and the social graph queries-per-second related to this application decreased by a factor of 10. Further, the time from when a comment was created on the device to the time it became visible on other devices decreased by a factor of two. (Data collected from our production environment is presented in §5.)

Unique aspects of Bladerunner. Four aspects of Bladerunner's design are unique and worth highlighting. First, Bladerunner's dissemination of updates is designed to be best-effort only, which leads to a simpler and more efficient overall design. Imposing Bladerunner to be reliable would in part require replicating the updates being disseminated across regions, which would add considerable overheads and attendant delays in getting the updates to the devices. Instead, Bladerunner detects failures that might have caused an update to be dropped, and then reliably informs all affected parties, including affected end-devices, of the failure.

This approach effectively shifts the complexity and overhead of dealing with failures to the exception handling parts of the system. This strategy works because we can assume (*i*) backend communication and services exhibit a baseline of reliability; (*ii*) TAO already stores all social graph updates persistently and in a replicated form; and (*iii*) the affected application can always obtain the latest social graph data through a polling operation to recover from a failure if needed. The strategy does, however, require that all participating components, including the proxies through which streams are routed, cooperate in the signaling of failures.

A second unique aspect of Bladerunner's design is that each Bladerunner application has its own set of BRASSes, each with its own implementation. This separation of concerns simplifies BRASS programming. BRASS is serverless in the sense that a new instance is spooled up automatically whenever a stream request arrives at a designated host that doesn't already have a running BRASS instance for the target application. Each application will have multiple running BRASS instances distributed around the world.

A third unique aspect is that upon social graph mutations, the data involved in an update itself is not pushed to Pylon (and subsequently to BRASSes), but only a corresponding update event, along with metadata characterizing and identifying the update in TAO. Thus, when a BRASS receives an update event from Pylon, it first needs to query the WAS to obtain the data before the data can be pushed to client devices. This design choice may seem counter-intuitive, but it substantially reduces bandwidth usage over cross-region communication links, which are a limited resource. Each update to the social graph already incurs cross-region communication overhead as TAO replicates updates across regions, and having Bladerunner propagate the same updates across regions again more than doubles cross region bandwidth used for each update.

The overhead of the extra query is offset by the fact that queries for individual updates from BRASSes incur far lower overheads than polling queries. The former typically only involve point queries while the latter involve (higher overhead) range and possibly intersect queries. The former typically obtain data from one TAO shard, while the latter obtain data from many shards.[5] Further, before data is sent to a client device, it first needs to be privacy checked (e.g., to ensure a user doesn't receive data from blocked users). These privacy checks are complex and sensitive, and in our operating environment are only performed within the WAS. As a result, communication with the WAS is required for each update being sent to a device in any case.

The final aspect of Bladerunner's design worth highlighting is that Bladerunner uses a new application-level communication protocol called ***BURST***. BURST is a *Request-Stream* (as opposed to a Request-Response) protocol that supports

[5] Consider a *LiveVideoComments* query "*fetch all comments on live video V since timestamp X*" in a scenario with a high volume of new comments for *V*. *V*'s comments and the index to the comments cannot be located on one TAO shard because the shard would become far too hot for both reads and writes, and hence a bottleneck. Increasing the number of TAO replicas will not help mitigate the bottleneck, because of the high write volume. Thus, the index and the comments need to be partitioned across many, many TAO shards, all of which have to be accessed when compiling a response to the query. In contrast, a query for a specific comment from BRASS requires data from just one TAO shard.

long-running streams that span across multiple hops of (possibly) different network protocols. We found it necessary to create a new protocol because (*i*) no single network protocol is uniformly available across the diverse set of browsers and mobile devices of our users; and (*ii*) connections between user devices and BRASSes involve multiple hops: first to a Point of Presence (POP) at the edge, then to a reverse proxy at the edge of the target datacenter, before ending at a BRASS. One of the key features of BURST is that it reliably informs the end points, as well as the nodes in between, of any failure, enabling BRASSes to implement reliable in-order delivery of updates if they so desire (§4).

The rest of the paper describes Pylon (§3.1), BRASS (§3.2), and BURST (§3.5) in more detail. §4 describes how Bladerunner handles failures. §5 presents scale and performance metrics obtained from our production environment. We first further motivate the need for a system like Bladerunner.

2 Motivation

The impetus for seeking a more effective solution for disseminating relevant social graph updates to end-user devices was the *LiveVideoComments* application. It was the application that drove the design of Bladerunner, and it is still the most complex Bladerunner application in use today.

Our *LiveVideo* application may have over a hundred million users around the world watching a popular event live, such as a solar eclipse or live election results. The *LiveVideoComments* application allow users to post comments to the live video they are watching. A subset of these posted comments should be displayed to end-users in as close to real-time as possible. To increase user engagement, only those comments that are "relevant" to the user should be displayed. Doing so is complicated by the fact that as many as one million comments may be posted within a few seconds at peak. We note that predicting the rate at which comments for a video are posted is infeasible. Some reasonably popular videos result in almost no comments, while less popular videos may generate many.[6] Further, some video streams have very few comments for prolonged periods of time, but then incur a burst of many comments; e.g., a video of a lunar eclipse when the eclipse happens.

Processing posted comments to determine which ones to show to which viewers is non-trivial. Many comments are spam, irrelevant, or generally of low quality; these need to be filtered out for all users. Comments older than *n* seconds become irrelevant and can be discarded. Comments in a language foreign to the end user need to be filtered out. Comments posted by users blocked by the viewer also must be filtered out. Comments posted by users the viewer does not know are less meaningful and should obtain a low rank (unless perhaps the commenter is a celebrity). Because a user cannot ingest more than one comment every second or two, the remaining comments need to be ranked so that at most one highly ranked comment is pushed to the viewer per second. The key here is that the stream of comments related to a live video needs to be filtered and rate limited on a per-viewer basis. (Other applications will have their own, different set of requirements for processing updates.)

[6] E.g., a live video of a cake baking can (surprisingly) be more popular than a streamed live presentation by the leading presidential candidate, yet the former garners very few comments while the latter generates many.

Table 1. Distribution of number of updates within a 24h period to targetted areas of interest in the social graph.

% areas:	83%	16%	0.95%	0.049%	0.0001%
updates:	0	< 10	< 100	> 1M	> 100M

We briefly review several different architectures we either deployed or experimented with to target the *LiveVideoComments* application.

Client-side polling: As described §1, this is the simplest solution but exhibits unacceptably high bandwidth consumption at the last mile to the device, excessive device processing overhead, and high querying load on TAO at the back end. This is primarily due to the fact that most polled queries return no new data. To illustrate this, we measured all areas of the social graph that client applications had expressed interest in receiving updates from in our current production system, to see how many updates to those occurred within a day. Tbl. 1 shows that the Pareto principle applies: roughly 80% of the areas have zero updates over a 24hr period, while a few selected areas have very high update rates.

Sharding and replicated caching: Backend querying bottlenecks are typically mitigated through sharding, replication, and distributed in-memory caches, as is done in TAO [13]. This distributes the querying load and allows queries to be serviced from a region close to the end-user. But this also comes at a price in that the number of copies of data being accessed needs to be large enough to accommodate peak load. Moreover, with in-memory caching, new updates have to be *reliably* pushed to the caches; otherwise, each query to the cache would need to query backend storage to check whether newer data is available.

Server-side polling: To reduce device and last-mile communication overhead, a simple improvement to client-side polling is to have a server-side agent, acting on behalf of the client, perform all polling. The agent pushes updates, when they become available, to the client. The client maintains a persistent connection to the agent, and re-establishes the connection whenever it fails. Server-side polling substantially reduces client and last-mile network overheads. But it still causes excessive backend server overhead for parsing, evaluating, and executing each incoming query poll.

Overall, polling in the above approaches is generally wasteful at the backend since the majority of polls come up empty.

Having operated *Messenger* using both (*i*) polling with sharding and caching and (*ii*) a push message delivery implementation, we found the polling solution needed eight times the hardware to achieve similar performance.

Distributed event logging: An alternative to polling TAO is to make use of a replicated and distributed event logging system, such as Kafka [6, 21, 31] or Pulsar [8]. Updates to the social graph are simultaneously published to the log as an event. Each different area of the social graph of interest is assigned a unique topic, and updates are reliably published to a topic with copies made across replicas. Clients regularly poll the logging system for updates instead of polling TAO. These polls incur less overhead (e.g., no intersect queries) and the immutability of data appended to the log simplifies replication.

In practice, however, existing logging systems are not designed to accommodate 100 million plus queries per second on a single topic, as required by some of our applications. Kafka is a good case in point. It is widely used in many production settings, and it has been scaled (by LinkedIn) to be able to handle as many as seven trillion events a day, albeit with a customized implementation [33]. However, it is not suitable for many of our applications. Kafka's current structure precludes it from supporting billions of topics that are created dynamically; e.g., LinkedIn's variant supports only 100,000 topics. Further, the primary mechanism for making Kafka scalable is partitions with events of a topic distributed across partitions spread across nodes in a cluster. Studies have indicated that Kafka does not scale well as the number of partitions increases beyond 100 partitions per broker; e.g., [50].[7] Worse, each event is assigned to exactly one partition, causing all accesses to an event to effectively be serialized.

Pub/Sub triggering: As an alternative to repetitive polling, a triggering solution uses a publish/subscribe (pub/sub) system [19, 51] to notify the client that an update of interest has occurred, and only then does the client poll TAO. This approach is similar to the one used by Thialfi [2]. It substantially reduces the volume of polling, since polls that return no data are eliminated. Publishing would entail first writing the message to TAO and then publishing a notification that an update has occurred, which in turn triggers a poll. However, the pub/sub system would need to guarantee at-least-once delivery of the notification to prevent clients from missing some updates, which implies that notification events would have to be replicated (across regions). The downside of this solution is that devices could easily be overwhelmed with update signals in some scenarios. Moreover, the triggered poll would still be subject to the latency added by having to use indexing in TAO.

[7] In fact, the current recommendation is to limit the number of partitions to 4,000 per broker and 200,000 per cluster.

Pub/sub data distribution: Applications like *LiveVideoComments* naturally map onto the topic-based pub/sub paradigm [19, 51]. With this approach, the pub/sub system is used to push updates directly to devices using a topic to identify the area of the social graph the update occurred in. Devices subscribe to the topics of interest to receive targeted updates.

However, almost all of the pub/sub systems that have gained traction are designed for backend operations that offer reliable delivery — e.g., Wormhole [49]. They are not well equipped to handle unreliable connectivity to mobile devices. Further, using a pub/sub system to push updates directly to devices will, in our environment, result in devices receiving a firehose of data on occasion, overwhelming the device and the last-mile connection. Finally, it is questionable whether the existing pub/sub systems are scalable so as to be able to handle the number and frequency of updates we needed.

Some pub/sub systems offer various forms of broker-side filtering to reduce the volume of data pushed to devices. However, we have found the available filtering capabilities far too limiting for more complex applications such as *LiveVideoComments*, since they typically only offer a limited set of filtering operations that can be combined using AND/OR boolean connectives. We spent years trying to implement our own generic pub/sub system with more complex server-side processing capabilities (for user-specific filtering, ranking, and privacy checking), but ultimately declared this initiative to be a failure. We found that we had to add more and more configuration parameters as new applications were onboarded, causing the space of configuration parameters to grow exponentially (something also observed in other domains [56].) It became increasingly difficult to reason about and configure applications correctly, in part because there were implicit orderings for applying the configuration parameters. The system, over time, became increasingly brittle, unwieldy, and unmaintainable. Exacerbating the situtation was the fact that some configuration parameters had complex interactions. Consider the interaction between rate-limiting and privacy checks. Checking privacy on each message adds unnecessary overhead, so it desirable to check only those messages selected for delivery. But with privacy checking after rate-limiting, the end-user may get fewer messages than intended.

Our experiences with the above approaches led us to design a new system from scratch, albeit using elements from the solutions described above.

3 Bladerunner design details

Recall from §1 and Fig. 2 the three server components involved: Web application servers (WASes), Pylon and BRASS. Typically, each update to the social graph is sent from an app on a client device to a WAS, that in turn (*i*) issues an

update request to TAO, and (*ii*) publishes the update to a "topic" through Pylon. Pylon, in turn, pushes the updates to BRASSes that have subscribed to the topic. Each area of the social graph that might be of interest is mapped to its own topic. Topics may be arbitrary strings, but in our domain are structured similarly to file names; e.g., `/LVC/videoID` for comments related to a live video, `/TI/threadId/uid` for information on whether a user is currently typing, or `/Status/uid` for information on whether a friend is currently online.

BRASSes subscribe to these topics with Pylon depending on the current interest of their device clients. When a BRASS receives published updates from Pylon, it processes the updates and pushes selected updates to the client devices. In doing so, BRASS filters, ranks, privacy checks, and applies other varied business logic on the updates it receives on a per client basis to determine which data and when to push to the client.

An application on a client device that wishes to receive updates, expresses its interest by issuing (for example) a GraphQL subscription request to a BRASS, which is translated to a topic. If successful, a *request-stream* is established between client device and the target BRASS. In practice, an application will have multiple (10+) active request-streams simultaneously. Each request-stream is routed independently and can fail independently. The BURST application-level protocol is used to support request-streams.

In the subsections that follow, we describe Pylon, BRASSes, and BURST in more detail, including how they are implemented.

3.1 Pylon

Pylon is a simple pub/sub system used to deliver published messages to BRASSes. It has two primary functions: (*i*) keeping track of BRASS subscriptions, and (*ii*) streaming all content published to a topic to all BRASSes subscribed to that topic. Pylon is content-agnostic and does not process published messages in any way beyond providing low-latency fan-out of messages to its BRASS subscribers. Pylon is designed to be as simple as possible so as to be highly efficient. It does not offer any guarantees on message delivery on failures (see §4).

For each topic, Pylon maintains a list of BRASS subscribers in a distributed in-memory key-value store that uses replication for durability. Rendezvous hashing on the topic is used to identify the KV stores used to maintain the subscriber information. The replicas are set up so that one is in the local region and the others are in different remote regions. This allows the service to be partially available during a partitioning event. A notable insight for Pylon was to decouple the status of the subscription information from the availability of data. From a CAP Theory point of view [12], the replication of the subscription information is designed to be CP, while delivery is designed to be AP, which enables applications to recover from failures in a graceful way if they care.

On receiving a publish request with a social graph update event, the target Pylon server first queries the appropriate KV stores to obtain a list of BRASS subscribers for the target topic, and then forwards the message to each BRASS host subscribed. For improved response time, Pylon initiates the forwarding of a published message when it receives the topic's subscriber list from the first-responding storage replica (typically in the local region). When the lists from the other replicas arrive, Pylon forwards the message to the subscribers that were not included in the subscriber list returned by the earlier replicas. If Pylon identifies inconsistencies in the subscriber information received from the replicas, it performs patch operations based on a quorum of responses to achieve eventual consistency on each Pylon storage node.

Our production environment operates with several thousand Pylon servers, spread across multiple regions. Topics are partitioned across 512K shards that are then mapped onto the physical servers. Mapping multiple shards onto one server allows incremental load rebalancing, one shard at a time. The number of requests processed per second is kept relatively low so there is headroom to absorb spikes when needed.

3.2 BRASS

BRASS instances are responsible for application-specific stream processing. Each instance is dedicated to a specific application (e.g., *LiveVideoComments*, *TypingIndicator*, etc.), but multiple instances per application are used for horizontal scalability. Each BRASS runs in its own isolated environment (e.g., VM).

Using a separate BRASS for each application offers several advantages: (*i*) the implementation becomes simpler because each BRASS addresses the requirements of only one application; (*ii*) debugging becomes easier, as only one application is involved; and (*iii*) the BRASS code becomes the vast majority of the configuration, so configuration-hell is avoided. Per-application BRASS processing also provides operational benefits when each instance runs in its own isolated environment, as a misbehaving application can be isolated and dealt with, without affecting the rest of the ecosystem.

BRASS is a general-purpose compute environment, allowing any arbitrary form of processing; in particular, it may invoke any backend service. We happen to use JavaScript V8 engines for our BRASS instances, each offering a single-threaded runtime environment similar to node.js. The JS VM follows a traditional asynchronous environment where all computation is powered by an event loop, executing logic on each incoming (e.g., device subscribe or Pylon publish) request and each backend service response.

When a new request-stream is being instantiated, the BRASS will issue a subscription request with Pylon for the

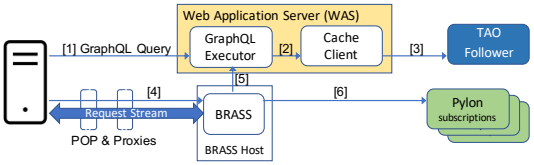

Figure 3. Data fetch and stream establishment.

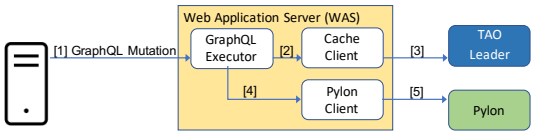

Figure 4. Mutation issue and publish.

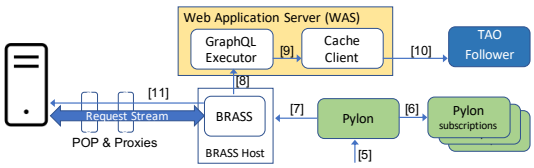

Figure 5. Update event fanout and data delivery.

appropriate topic if it is not already subscribed. When a social graph update event is received, the BRASS will fetch the GraphQL data from a WAS when needed. Many BRASS implementations are stateful so they can serve their clients in a more meaningful way, but that state is ephemeral and lost on a BRASS failure.[8]

We stack multiple BRASS applications on the same host using multi-tenant design principles, allowing us to accommodate diverse application load requirements. Several thousand hosts are dedicated to run BRASSes in our datacenters. Each BRASS runs in its own VM,[9] and in the common case, the number of BRASSes per host is limited to two per core to reduce context switching. The number of BRASSes instantiated across different machines for a given application is partly based on (memory and CPU) load, and can be scaled up or down as needed, but locality may also be taken into account. Some applications have predictable flows, while others have spiky flows. Those with predictable flows are well-suited for multi-tenant colocation, while those with spiky flows are best isolated.

Proxies determine which BRASS host to route device subscription requests to. This routing is based on load, topic, or a combination of both, depending on application configurations. For applications with low fanout (i.e., few clients), routing is typically based on topic, so as to curtail the number of subscriptions maintained by Pylon, while applications with high fanout are typically routed based on load. BRASSes can also be configured to run on dedicated hosts for physical isolation, or on any host with available capacity. The configuration information is maintained in ZooKeeper [28].

3.3 Query and data flows

We briefly outline the high level control and data flow of Bladerunner's operation.

Data fetch and stream establishment: (Fig. 3)

1. To obtain current TAO state, the client device issues an initial GraphQL query to a WAS in a request/response fashion (HTTP GET).
2. GraphQL Executor on the WAS issues read request(s) to a TAO cache client.
3. Cache client issues reads from a TAO follower.
4. Afer receiving a response to the GraphQL query, the client device renders its UI and asynchronously issues a GraphQL subscription request to a BRASS.
5. BRASS calls WAS to resolve subscription request into a concrete topic.
6. BRASS registers the BRASS host as a subscriber to the topic with Pylon.[10]

These steps, if successful, will establish a bi-directional request-stream between device and BRASS, and BRASS can now push GraphQL payloads to the device at any time.

Mutation issue and publish: (Fig. 4)

1. Client device issues a GraphQL mutation to a WAS in a request/response fashion (HTTP POST).
2. A GraphQL Executor on the WAS looks up the implementation of the mutation field and converts the mutation into a TAO write.
3. The Cache Client calls the appropriate TAO node, which commits the write.
4. An update event is emitted based on business logic and published to local Pylon client using the appropriate topic. The event may include metadata such as uid, quality score, etc.
5. The Pylon Client library in the WAS issues a publish request to the appropriate Pylon server.

Update event fanout and data delivery. (Fig. 5)
With each publish request received from WAS ([5]):

6. Pylon fetches a list of BRASS hosts that have subscribed to the topic.
7. Pylon fans out the update event to all target BRASS hosts. On each BRASS host, a "subscription manager" fans out the event to all subscribed BRASSes. BRASS buffers and filters events on a per-device basis based on event metadata.

[8] Given that BRASS is a general compute platform, the BRASS application can in theory make the stream data it is processing durable, but this is effectively never done. §3.5 and §4 describe how BRASSes can implement reliable delivery.

[9] As an exception, there is a long tail of very simple applications that are combined to share the same BRASS and VM.

[10] Each BRASS host runs a Pylon subscription manager. Pylon subscription requests from a BRASS are directed to this manager that only forwards the registration to Pylon if no BRASS on the same host has an existing subscription to the topic.

8. BRASS makes a call to the WAS, if needed, to issue a GraphQL query to obtain the GraphQL payload for the target update and to privacy check the query.
9. WAS fetches data from TAO cache.
10. On a miss, cache client reads from a TAO storage node.
11. BRASS sends update to device, if warranted, where product logic updates client-side state and triggers re-rendering of UI.

3.4 Sample BRASS implementations

We briefly describe four widely-used Bladerunner applications and how they are implemented. Recall that each of these applications is implemented and runs completely independently of the other applications. Each of the implementations requires at most a few hundred JS lines of BRASS code.

***LiveVideoComments*:** As described in §2, the objective is to allow clients to post comments on a live video. The most relevant comments are made available to the other viewers of the video at a prescribed maximum rate so the UIs can display them. Comment relevancy is determined on a per viewing client basis.

In a nominal implementation of this service, the WAS creates an update event for each posted comment and tags it with metadata, such as uid of the poster, quality score (generated by an ML algorithm), language, timestamp, etc., before publishing it to Pylon using the topic `/LVC/VideoID`, where `VideoID` identifies the video. Each device displaying the video subscribes to this topic with a BRASS (that in turn subscribes with Pylon if not already subscribed). Each *LiveVideoComments* BRASS maintains a ranked buffer for each stream-connected device to which it adds the incoming updates after filtering them on a per user basis. For the most relevant ones, BRASS fetches the comments from the WAS. The highest-ranked comment in the buffer is pushed to the device periodically at a prescribed rate.

This straightforward implementation does not scale to the level we need for highly popular videos where a million comments may be posted within a few seconds. To handle this type of load, BRASS and the WAS switch strategy. The WAS pre-ranks posted comments, discards low-ranked comments, publishes extremely high-ranked comments as update events to topic `/LVC/VideoID`, and publishes the remaining comments as update events to topic `/LVC/VideoID/uid`, where `uid` identifies the user who posted the comment. To populate the ranked buffers, BRASS subscribes to `/LVC/VideoID` as well to `/LVC/VideoID/a-uid` for each friend of each stream-connected video viewer.

***ActiveStatus*:** The objective is to display the online status of a user's friends.

Each device updates the client's status to `ONLINE` with the WAS every 30 seconds when online. WAS publishes this update event to Pylon using topic `/AS/uid`, where `uid` is the id of the client. When a device stream-connects to a BRASS, the BRASS subscribes to topic `/AS/f-uid` with Pylon for each of the user's friends, where `f-uid` is the id of the friend. (Thus one device subscribe results in many BRASS subscriptions.) When BRASS receives an update event from Pylon, it updates a map of online friends for each stream-connected client (taking a 30 second TTL into account), and periodically pushes a batch update to the device. Pushing batches only periodically prevents the device from receiving too many updates.

***TypingIndicator*:** The objective is to display dancing ellipses when a communicating counterparty is typing.

When a user starts/stops typing, the device application issues a typing-indicator update to WAS, which publishes the event to Pylon using topic `/TI/threadId/uid`, with `threadId` identifying the communication thread and `uid` the user. The device also subscribes to topic `/TI/threadId/-c-uid` where `c-uid` is the id of the communicating counterparty. Update events are pushed to the device as they arrive.

***Stories*:** The objective is to allow users to highlight photos, video clips, or other content to their friends. Stories are organized into "containers," with each container comprising stories of one user. The stories are removed after a set period of time (e.g., 24 hours). Each user's UI displays thumbnails of the n highest-ranked containers of their friends.

When a user creates a new story, the WAS publishes it to Pylon, which in turn pushes it to subscribed BRASSes. The publication includes metadata that identifies the user. Each BRASS maintains, for each connected device, a list of containers with stories from the user's friends, ordered by rank (as determined by a ranking algorithm). BRASS then pushes (i) new stories that should be added to the device's existing containers, (ii) containers that have become ranked high enough so they should be displayed, and (iii) container deletion requests. Thus, the BRASS effectively manages what is being displayed on the device. We note that with polling, two intersect queries (with relatively high TAO overheads) are required to obtain the n highest-rank containers. Thus, this Bladerunner application substantially reduces client device polling, because only one poll is needed when the application commences, yet the displayed containers remain up-to-date.

3.5 BURST

BURST (Bladerunner Unified Request Stream Transport) is a new application-layer protocol developed for Bladerunner to support long-running request-streams that connect client device applications and BRASSes. BURST objectives are to (i) provide a uniform networking API to Bladerunner applications; (ii) implement request-streams as first-class citizens where each stream is routed independently across multiple hops and can fail independently; and (iii) simplify how Bladerunner applications handle failures.

Regarding (i), we note that no individual available networking API (e.g., WebTransport [53], WebSocket [39], XMLHttpRequest [55], or Fetch [54]) is supported across all browsers. Mobile devices, on the other hand, will support TCP and

QUIC [32], or one of the protocols built on top of them (e.g., MQTT [27], HTTP/2, HTTP/3, or WebSocket). However, using a separate TCP connection per request-stream is too heavy-weight. HTTP would be better suited and is the only protocol we are aware of that has historically worked well across multiple network hops with clear semantics for the roles of proxies. Unfortunately, some client devices do not support UDP so HTTP/3 cannot be used across the board, leaving HTTP/2 as a fall back.

Regarding (*ii*), each request-stream between device and BRASS needs to be routed independently across at least two hops: first through a POP (Point-of-Presence) access point and then through a reverse proxy layer at the BRASS data center. We note that the existing stream protocols all have serious drawbacks. *HTTP/SSE* is a streaming protocol, but is unidirectional so it has no way to pass information to the server. *MQTT*, while specifically designed for pub/sub systems supporting lightweight IoT edge devices, is a session-oriented, point-to-point protocol where load balancing and failure handling are performed on a per session basis, not on a per-stream basis. *RSocket*, supports request-stream and bidirectional stream communication [46], but load balancing is the responsibility of the client, and flow control is based on number of messages, not bytes, making it challenging when messages have highly diverse sizes.

BURST offers a uniform request-stream abstraction to its applications. Each stream is routed independently between the client device and the handling BRASS instance (but routing is sticky, unless rerouting is needed for load rebalancing or failure recovery). On each network hop, multiple streams are multiplexed onto the underlying network protocol used for transport. The core stream transport guarantee offered by BURST mirrors the guarantees offered by TCP: (1) messages sent over a stream arrive in order, and (2) failures are signalled to the participating nodes (analogous to TCP failures being signalled through socket disconnects). However, because a stream spans multiple participants with routing responsibilities, failure signalling is more differentiated than just a socket disconnect, as described further below.

Without delving into details, BURST messages from client applications support subscribe requests to instantiate a new request stream, cancel requests to terminate a stream, and acknowledgements. A subscription request includes (*i*) a `sid`, a unique id generated by the client, (*ii*) a `header` used to indicate the properties of the request, including the GraphQL subscription; and (*iii*) an optional body which is an opaque binary blob of data that only the target BRASS needs to understand. We happen to have standardized on a JSON format for the header that may include fields, for example, to inform BRASS to connect to a different data source because developers are testing a new feature, or to express client versioning so BRASS can understand the limits of the client.

Upon the establishment of a stream, the associated BRASS will issue a sequence of stream responses. Each response contains a batch of what we call *deltas*. The batch is transmitted atomically in one unit (for networking efficiency), but is also processed client side atomically (in an all or nothing fashion). A batch may include zero or more social graph updates as deltas. Other types of deltas include stream terminations and flow-status messages to signal failures as described below.

POPs and proxies are central participants in the protocol. The subscription `header` is visible and interpreted by the proxies as the it passes through, and used for routing purposes. The proxies keep a copy of the current header and body of each stream passing through them so they can take recovery action when failures are detected (§4). (They garbage collect this stream state when the stream is explicitly terminated or when the connection to the device fails.)

Unique BURST features: Two features set BURST apart from other existing streaming protocols. First, BURST guarantees that both failures and recovery from failures are signalled to all participants of the affected streams as they occur. This is important so that the participants are aware of the status of the streams in a timely fashion, given that a stream may have no messages for prolonged periods of time. For example, it allows the client app to display connectivity issues to the user, and it allows proxies and BRASS to garbage collect stream state after a timeout. The signaling occurs even if the failure was temporary and quickly recovered from (leaving the stream in tact). A `flow_status` delta is used for this signalling. How it is used is described in §4.

The second unique feature of BURST is a mechanism that allows BRASS to control the stream state used by the client and intermediary proxies to reconnect after a failure. Generally the state is a copy of the subscription request, including the header and optional body. By sending the client a `rewrite_request` delta, BRASS is able to modify this state at any time and override the existing header with a new one. BRASS's ability to rewrite the client state enables a number of interesting features. We highlight three of them.

Sticky routing. When the connection to a device is dropped, it is desirable for the device to reconnect each of its streams to the same BRASS instance that previously served the stream (e.g., to benefit from stream state not yet garbage-collected or from improved cache hit rates). Bladerunner achieves this by having the BRASS issue a rewrite request to patch the request headers with information identifying the BRASS as soon as a stream is instantiated. Then, when a device re-subscribes after a failure, it will use this re-written header so that the subscribe request lands on the BRASS that previously serviced it. Without rewrites, the (device, stream) to BRASS mapping would have to be maintained explicitly by the device or stored in some datastore.

Resumption. Some applications assign sequence numbers to updates for a device so that when the device re-connects after a dropped connection, it can identify the last sequence

number it received and only receive subsequent updates. With rewrites, the servicing BRASS can patch the header to include the sequence number of the last update sent to the device. Then when the client resubscribes after a failure, the subscribe request will automatically identify the last message that was received. This simplifies client logic because the sequence numbers don't have to be tracked explicitly. More generally, BURST's rewrite mechanism enables BRASS to leverage any reliable messaging stack that depends on sync tokens, sequencers, and other state. For example, the state of a rate limiter can be stored in the header so that when a BRASS failure occurs, the resubscribe will include this information and the new servicing BRASS can take this state into account.

Redirects. A BRASS may decide that a stream is to be redirected to another BRASS, say for load balancing, BRASS consolidation, or because the host on which the BRASS is running is to be taken offline. This can be achieved by having the current BRASS issue a rewrite request that patches new routing information into the headers, and then terminate the stream thus informing the device to retry.

All of these behaviors could also be achieved by having the client maintain the individual states separately and have BRASS send the client metadata that causes the state to be updated on a state by state basis. However, rewrites offer a general solution so that the client need not be aware of the states, and in fact allows the introduction of new state transparent to the client. The latter is particularly important because mobile client software is often not updated for long periods of time [45] and not conducive to continuous deployment (in contrast to server software [47]).

4 Failure handling

Bladerunner delivers updates to client devices on a best effort basis only. However, it goes to great effort to communicate to all affected parties when it detects a failure that might have caused an update message to be dropped. Fundamentally, it is up to application logic to recover from messages that were not reliably delivered, if the application so desires. In practice, most of our applications can tolerate dropped updates without negatively impacting the user experience, if they occur infrequently; e.g., occasionally incorrect typing indicators, friend status, or a missed live video comment.

While many aspects of failure recovery are omitted here due to space limits, we briefly outline three *axioms* that govern how failures are handled in Bladerunner. The first axiom regards ***failure notification***: if a component detects a failure, then it informs the (upstream and downstream) components to which it is still connected, and this information is further propagated to the affected stream end points — either client devices (if downstream) or BRASSes (if upstream).[11]

For example, if POP P_i fails, this will be detected by the connected devices and each proxy P_j to which P_i was connected. The devices inform the affected apps and the P_js inform the affected BRASSes. This allows the device app, for example, to recover by issuing a WAS poll to get the latest state and then reconnect to a BRASS. Or it may decide recovery is not needed and simply resubscribe.

If a client device fails or loses TCP connectivity, POP P_i will detect this, and it will inform all BRASSes servicing streams instantiated by the device. If a BRASS host fails, P_j will detect this and inform all devices whose streams were being serviced by the host; Pylon also detects this and removes all subscriptions from that host. Finally, if a Pylon server fails such that a quorum is lost, then the connected BRASSes will detect this and inform all their clients.

The second axiom regards ***connectivity recovery***: the component downstream from a failure that is closest to the failure will re-establish connectivity and repair each affected stream. To allow this, proxies keep a copy of the most recent subscription request (potentially modified via a rewrite) of each stream, which they use to resubscribe clients. The recovered streams may now be routed through a different POP or proxy, or be routed to a different BRASS. If routed to the same BRASS, we note that the BRASS has the option to keep the state of failed streams for a period of time to make reconnections more seamless. Once a stream has been re-established, BRASS informs the device of this, and the device decides how to recover from the fact that it may have missed some updates.

For example, if a device detects that the connection to POP P_i has been dropped because P_i failed, it will re-establish a connection to an alternative POP and then resubscribe to the topic for each affected stream. In doing so, it will use the currently stored subscription requests which, in the common case, go to the same BRASSes that previously served the streams. If proxy P_j fails, each affected POP P_i will re-establish a TCP connection to an alternate proxy P'_j, resubscribe to the topic on behalf of each affected client stream using the header last rewritten by BRASS to the client.[12] If a BRASS fails, then affected proxies, P_js, connect to alternate BRASS instances, and if a Pylon server fails, then BRASS connects to an alternate Pylon server instance.

The third axiom regards ***stream state recovery***; BRASSes are generally responsible for the recovery of the stream state if needed by the application. (As we noted earlier, retaining stream state with its attendant overhead is not critical for the vast majority of our use cases; e.g., applications with a

[11] One of the challenges is detecting failures in a timely fashion. For example, waiting for TCP to signal a failure may take too long. We employ a number of techniques to detect such failures more quickly; e.g., by using heartbeats.
[12] Proxy handling of failures may be application-dependent.

BRASS rate limiter may see a slight burst of message deliveries to the client.) If the retention of stream state is needed, then the state can either be persisted in the stream (through rewrites) or on a backend store. In both cases, the state can be recovered after a BRASS failure.

In effect, Bladerunner's design objective with respect to failure handling is to have the system deliver updates quickly, albeit unreliably, and instead enable applications to implement fault tolerance if they desire. It is based on the fact that failures at the backend (within datacenters) are exceedingly rare (as shown in §5) and most failures occur at the last mile. For those applications that cannot tolerate dropped updates, the applications can implement the reliability they need and in their own way.

Application-specific logic is implemented within three components: the WAS, the BRASS, and the device. Together, they are responsible for implementing the level of fault tolerance needed by their application. One approach is to simply have the device reissue a poll and resubscribe whenever it detects that a failure that affected the stream has occurred. Another approach is to have the WAS add sequence numbers to the updates sent out to Pylon, and if, for example, BRASS detects missing updates, it can query the WAS for the missing data. Further, BRASS can rely on device acks to ensure the device receives each update.

As an example, consider *Messenger* that requires reliable delivery of messages. On the backend, each user has a mailbox, organized as a list of communication threads that in turn have a list of messages ordered by time. If a thread has, say, five users, then each new message to the thread will be seperately added to all five mailboxes. Each time a message is added to a mailbox, it is assigned the next consecutive sequence number for the mailbox. This allows dropped messages to be detected both at the BRASS and at the device, although BRASS will recover the dropped message so the device does not have to. If the connection to the device fails, the device will resubscribe with the latest sequence number it obtained, at which point the BRASS polls the mailbox to obtain all subsequent messages.

5 Metrics from our production deployment

In this section we present Bladerunner metrics that were obtained from our live production environment. We first describe observed effects of switching the *LiveVideoComments* application from a polling solution to Bladerunner at the time the switchover occurred. We then characterize the current Bladerunner workload and quantify the scale at which Bladerunner operates. We also identify the magnitude of failures that Bladerunner must cope with.

Polling vs. Bladerunner metrics. When *LiveVideoComments* switched from a polling solution to Bladerunner, various metrics were collected to be able to compare the two approaches. Fig. 6 depicts the distribution of latencies for

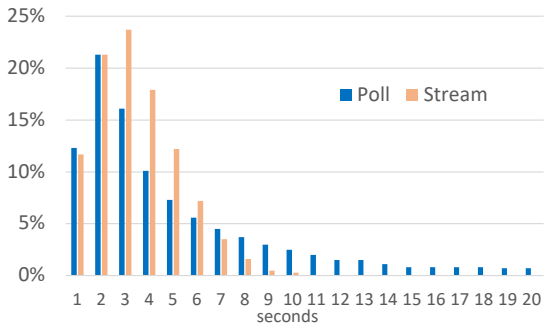

Figure 6. Latency distribution for *LiveVideoComments*

Table 2. Request-stream lifetime distribution.

<15 min	15min–1hr	1hr–24h	24hr+
45%	26%	25%	4%

LiveVideoComments from when comments were posted to when they became available at the edge. The key observation is that polling exhibits a long tail in latencies while Bladerunner does not. Eliminating this long tail stabilized the expected latency from 4.8 seconds to 3.4 seconds, reduced p75 latency from 6 seconds to 4 seconds, and reduced p95 latency from 14 seconds to 6 seconds, on average. Further, Bladerunner was able to better fit the product requirements by buffering comments up to a maximum of 10 seconds so it only pushes the most relevant comments. (Note that the choice of 10 seconds here was made by the product team as a reasonable tradeoff between timeliness and displayed comment relevancy for this particular application.)

When switching over to Bladerunner we also observed a 0.7% reduction in TAO queries, and a 0.4% reduction of CPU load at the Web application tier on average. This may not sound like much, but translates into a savings of thousands of hosts. More dramatic, on peak loads, we observed up to a 5% reduction on global IOPS on TAO, and a 2.5% reduction of CPU load on the WASes.[13] There are two reasons for these reductions. First, duplicate comment queries per viewer are eliminated with Bladerunner. Second, the pressure on the graph index which binds comments to a video was substantially reduced. While Bladerunner still requires BRASS to query WAS for each comment and user, these queries are point queries with good caching characteristics and hence straightforward to scale; in contrast, polling queries involve range and intersect queries (e.g., to fetch comments from a user's acquaintances), and the indices required for these queries do not easily scale when rapidly changing.

Workload characterization. Devices running Bladerunner applications typically have request-stream connections that number in the tens. Recent data shows each browser

[13] The factor 10 decrease in number of polls mentioned in the §1 was specific to the *LiveVideoComments* application, whereas the reductions here are relative to all traffic.

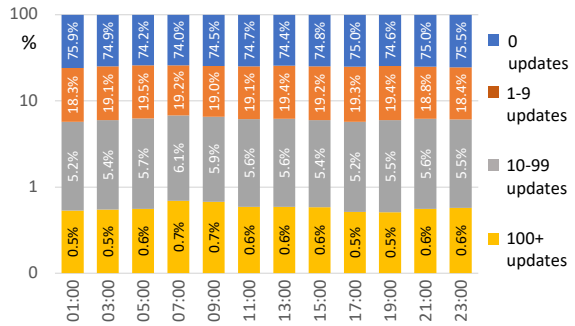

Figure 7. Percentage of request-stream subscriptions with 0, 1–9, 10–99, and over 100 publications. Note the log scale of the y axis.

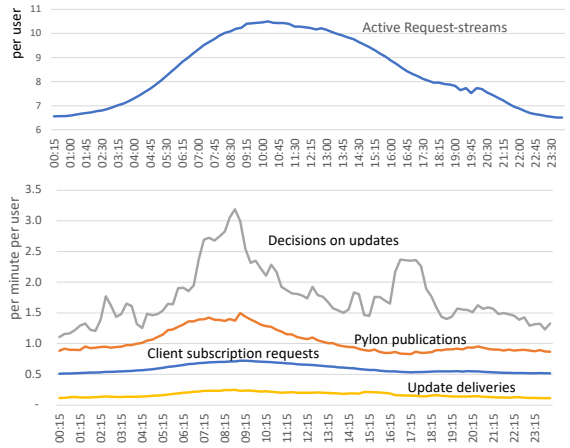

Figure 8. Per-user metrics from our production environment for a typical day. Each data point represents a 15 minutes interval and shown as the average of 15 measurements, one taken for each minute.

tab can have up to 60 concurrent streams and each mobile app up to 20.

The metrics that follow were obtained on a typical day in March 2020. Figure 7 shows per request-stream update activity. We selected twelve points in time, two hours apart, identified the request-streams that were active at that time, and counted the total number of update events targeting each request-stream's subscription during the stream's *entire* lifetime. The graph shows that 75% of the client subscriptions involve no updates at all. Fewer than 0.7% have 100 or more updates. These numbers support the thesis that any solution based on polling would be wasteful. Table 2 characterizes the lifetime of request-streams: 45% last less than 15 minutes and over 70% last less than one hour.

Bladerunner scalability. Fig. 8 shows Bladerunner activity over the course of 24 hours. The numbers presented are normalized to be on a per-user basis, whether online or not. The top curve shows that the total number of active Bladerunner request-streams follows a diurnal pattern. Each data point represents a 15 minutes interval and shows the average of 15 measurements, one taken for each minute. The average number of device subscription requests per minute

Table 3. Latency of Bladerunner sub-operations. All times in milliseconds. WAS: Web Application server. *LVC*: *LiveVideoComments*. Data shown are average times, as obtained from sampling 0.1% of all operations over a period of one week. Pylon latencies are shown separately for streams with fewer than 1,000 subscribers, between 1,000 and 10,000 subscribers, and streams with more than 10,000 subscribers. The BRASS overhead shown are for those applications that do not buffer and rank updates. Of the 76ms, 60ms is used to query WAS and the rest is for BRASS processing.

WAS receives update request	*LVC*	2,000
→ request sent to Pylon	other	240
Pylon receives update publish request	subscribers	
→ update sent to *n* BRASSes	<10,000	100
	≥ 10,000	109
BRASS receives update request		
→ sent to devices		76
Subscription request arrives at Gateway		
→ replicated onto Pylon		73

varies between 0.5 and 0.75 per user, which corresponds to slightly over 5,000 subscriptions per second per BRASS host. At peak, we have measured 3B device subscriptions per minute. Request-stream terminations are not shown but complement the subscription requests to arrive at the total number of active streams. The average number of publications emitted from Pylon per minute varies between 0.8 and 1.5 per user. Publications cause the BRASSes, in aggregate, to make between 1.1 to 3.2 decisions per minute per user to determine whether or not to deliver a message to a device. The number of decisions tends to be more variable as it is based on the particular workload active at the time; moreover some applications can exhibit high volatility; e.g., *LiveVideoComments* has an almost steady state of 25M decisions per minute but can quickly jump to 4B decisions per minute. Finally, the number of updates delivered to devices varies between 0.1 and 0.25 per minute per user on average. At peak, we have measured well over 1B update deliveries per minute.

While most request-streams receive no updates, we have observed that when streams become "hot," they can become exceptionally hot — to the point where many devices would have difficulty receiving and processing the high incoming flow. This justifies our decision to filter and rate-limit messages on the BRASS side. In effect, one of the primary responsibilities of BRASSes is to drop messages intelligently, as 80% of messages are filtered out at BRASS instances (1-(*deliveries*/*decisions*). Because effective per-stream filtering and rate limiting require application insight, they become easier to implement when done on a per-application basis. The number of client subscription requests and stream cancellations is substantial, justifying the need to make these as lightweight as possible.

Bladerunner Latencies. Table 3 presents the latencies incurred at individual Bladerunner components. They were

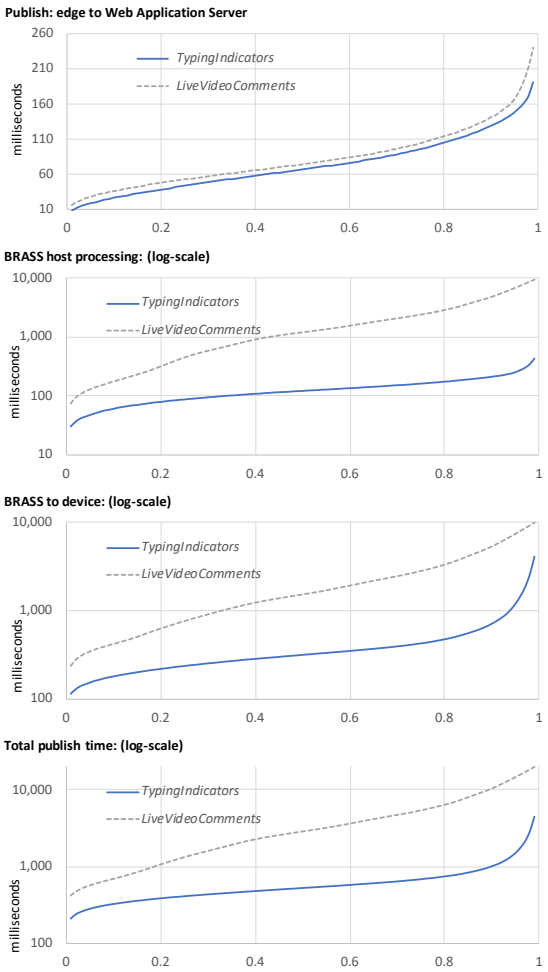

Figure 9. Cumulative Distribution of update latencies in milliseconds for two applications obtained from randomly sampling 100,000 updates to data related to these applications. Note the scale of the y axes are different in each of the graphs and that they don't start from zero. All graphs except the top one are in log scale.

obtained from system logs by sampling 0.1% of measured events over a period of one week. We note that the average latency for processing a *LiveVideoComments* update at WAS is roughly 2 seconds when measuring from the time the corresponding TAO mutation has completed to when the update has been sent to Pylon. On average, 1,790ms of this time is spent on ranking the quality of the comment, so only quality comments reach the BRASSes. Update requests that do not require ranking at the WAS take 240ms on average (as indicated by "other" in the table). The latency incurred at Pylon is reasonably modest: for streams with fewer than 10,000 device subscribers, the latency is 100ms on average with P90 and P99 percentile values at 160ms and 310ms, respectively. The latency of registering a BRASS subscription with Pylon is also modest when only including backend (datacenter) operations. However, the latency is considerably higher when measuring the latency from when the devices issues a

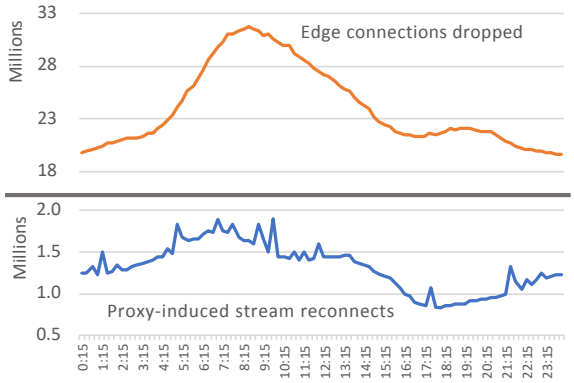

Figure 10. Top: number of last-mile connections dropped per minute. Bottom: stream reconnections per minute initiated by proxies. Each data point represents a 15 minutes interval and shown as the average of 15 measurements, one taken for each minute. Note that the y axis is different for the two graphs and neither starts at 0.

subscription request, primarily due to mobile network overheads: the average is 490ms (P90: 540ms) for subscribers in North America and Europe and 970ms (P90: 1,360ms) when measured for subscribers in all countries.

Fig. 9 depicts the cumulative distributions of update distribution latencies for *TypingIndicator* and *LiveVideoComments*, with clients located around the world. The metrics were obtained by randomly sampling 100K update events. The *TypingIndicator* application here is a generalized version of the one described in §3.4. Both applications require the BRASS application to perform privacy checking and device-specific transformations by making calls to backend services. The top three sets of graphs show different components of the total latencies (shown at the bottom): (*i*) latency of update requests from the edge proxy to a WAS, (*ii*) latency of request handling within a BRASS server which includes all interactions with Pylon and backend services as well as the batching of requests, and (*iii*) the latency for pushing the update to the device. *LiveVideoComments* rate limits each stream to one message every two seconds and the ranking is held fixed at 5 elements. Further, the latency of sending *LiveVideoComments* updates from the edge to devices is significantly higher than for the other applications because it competes with video bandwidth at the edge.

Failure handling. The top graph in Fig. 10 shows the number of last-mile connections to devices that are unintentionally dropped per minute. Each of these connections may have supported multiple streams. Thus, each BRASS host deals with thousands of disconnected steams each minute.

With respect to failures on the infrastructure side, the overwhelming majority of system events requiring a proxy to reconnect streams occur because of BRASS software upgrades and load rebalancing, with outright BRASS failures occurring very rarely. The bottom graph in Fig. 10 depicts the number of such proxy-induced stream reconnects. As a

point of comparison, from March 30 to April 5 there were a total of 33 failure events due to quorum breakage in Pylon.

6 Related Work

Over the years, numerous systems have been designed for disseminating data efficiently, often referred to as publish/-subscribe, reliable messaging queue/bus, stream processing, log/data collection/aggregation, event sourcing or log replication systems [2, 5, 9–11, 14–20, 25, 26, 30, 36–38, 41, 49, 57]. Cloud providers offer numerous such servies (e.g., [3, 4, 23, 34, 35], and there are open source solutions such as RabbitMQ [40] and ActiveMQ [7] as well as commercial offerings such as TIBCO's Rendezvous [52]. However, as outlined in §2, none of these systems are able to meet the needs of our applications in our target environment. They either do not scale sufficiently, are not able to offer the degree of user- and application-specific processing, or are ill-suited for dealing with unreliable last-mile connections.

Many of the existing pub/sub systems target in-datacenter server-to-server workloads where they can assume communication channels are robust. For example Facebook's *Wormhole* delivers 5 trillion messages a day [49], but it cannot cope with unreliable connectivity, and it cannot offer per-client message processing.[14] Google's *Thialfi* is a scalable pub/sub service that notifies clients when stored objects have changed so they can poll the backend store to obtain the latest version[2]. Thialfi's notifications are reliable with at least once delivery guarantees, but with attendant overheads for the common case, and they would still cause excessive range and intersect query overheads, given that Bladerunner's BRASSes filter out over 80% of all update events. Kafka, a scalable message bus with a pub/sub API developed by LinkedIn [6, 31] is able to process 7 trillion messages a day [33]. However, it is fundamentally a polling solution and would be inefficient for our workload for which most polls return no new data. Further, its structure precludes it from supporting billions of topics that are created dynamically — LinkedIn's variant customized for scalability supports 100,000 topics [33].

Ably [1], PubNub [42], and Pusher [43] are closer in spirit to Bladerunner as they target real-time server-to-device workloads. However, their internal architectures are not publicly known, and it is unclear whether they are anywhere close to Bladerunner's scalability.

Finally, Bladerunner is loosely related to serverless edge compute platforms like Cloudflare Workers [29] and Fastly Compute@Edge [22]. In contrast to these systems, Bladetunner focuses on supporting long running streams and chooses to operate BRASSes within datacenters instead of at POPs, primarily because it needs to perform many TAO queries on behalf of each stream-connected client.

[14] Wormhole supports a simple form of broker-side filtering that is far to limiting for our needs.

7 Concluding remarks

We described the architecture of Bladerunner, a framework for efficiently delivering relevant data mutations in backend datastores to user devices at the edge in a timely manner as the mutations occur. Bladerunner enables device UIs to display relevant up-to-date data. Datacenter-based BRASS stream servers play a key role in Bladerunner. They process streams of data updates on a per-application and per-user, customized basis. BRASSes significantly reduce querying load on backend datastores and the volume of data sent to devices. Bladerunner itself disseminates data updates on a best-effort basis. However, the fact that failures, which may have caused data to be dropped, are reliably detected, and the affected components are reliably informed, allows BRASS stream servers and devices to build a layer that offers reliability, if needed. Our experiences over five years operating Bladrunner in production have been positive; the trajectory of applications being onboarded continues to increase.

Portions of Bladerunner's design was influenced by the need to operate at very large scale, as evidenced in §5. This raises the question of whether it would be suitable for environments operating at lower scale. We argue that using serverless stream processors like BRASS, brings substantial benefits regardless of scale, as they significantly offload backend data servers, devices, and/or last-mile communication channels. However, two aspects of Bladerunner's design may be worthy of reconsideration when operating at lower scale. First, one might consider replacing Pylon (a best-effort pub/sub system designed for fast data dissemination to BRASS stream servers) with a logging system (e.g., Pulsar) that offers ordering and delivery guarantees, if it can support the scale needed. Second, one might consider using separate BRASS stream processors for each request-stream, instead of having BRASSes service thousands of streams. This would provide better isolation and further simplify the programming of these stream servers.

Acknowledgments

We attribute the success of Bladerunner to all current and past Bladerunner team members at Facebook. We would particularly like to thank Peter Weng, who pioneered the idea of offering real-time pub/sub as an infrastructure service, Narendra Parihar, who built and productionized Pylon, Sergey Volegov and Melissa Winstanley, who build and productionized BRASS (both the C++ service that hosts JS VMs and the JS framework for authoring applications), and Ravindra Rathi, Abhishek Srikanth and Hrishikesh Pathak, who built Request Stream end-to-end. We thank Kaushik Veeraraghavan and Jason Flinn who provided valuable feedback on an early version of this paper. Finally, we thank the anonymous reviewers and especially our shepherd, Oana Balmau, for valuable feedback on our submission — they have helped make this a far better paper.

References

[1] Ably. 2020. *The new standard in realtime edge messaging.*

[2] A. Adya, G. Cooper, D. Myers, and M. Piatek. 2011. Thialfi: A client notification service for Internet-scale applications. In *Proc. 23rd ACM Symp. on Operating Systems Principles*. 129–142.

[3] Amazon AWS. 2020. *Amazon Kinesis.* Retrieved Apr 2020 from https://aws.amazon.com/kinesis/

[4] Amazon AWS. 2020. *Amazon Simple Notification Service.* Retrieved Apr 2020 from https://aws.amazon.com/sns/

[5] Aleksandar Antonić, Martina Marjanović, Krešimir Pripužić, and Ivana Podnar Žarko. 2016. A mobile crowd sensing ecosystem enabled by CUPUS: Cloud-based publish/subscribe middleware for the Internet of Things. *Future Generation Computer Systems* 56 (2016), 607–622.

[6] Apache Software Foundation. 2017. *Apache Kafka: A distributed streaming platform.* Retrieved Apr 2020 from https://kafka.apache.org

[7] Apache Software Foundation. 2021. *ActiveMQ: Flexible & powerful open source multi-protocol messaging.* Retrieved Aug 2021 from https://activemq.apache.org

[8] Apache Software Foundation. 2021. *Apache Pulsar.* Retrieved Apr 2021 from https://pulsar.apache.org/en/

[9] Guruduth Banavar, Tushar Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. 1999. An efficient multicast protocol for content-based publish-subscribe systems. In *Proc. 19th IEEE Intl. Conf. on Distributed Computing Systems*. 262–272.

[10] Raphaël Barazzutti, Pascal Felber, Christof Fetzer, Emanuel Onica, Jean-François Pineau, Marcelo Pasin, Etienne Rivière, and Stefan Weigert. 2013. Streamhub: A massively parallel architecture for high-performance content-based publish/subscribe. In *Proc. 7th ACM Intl. Conf. on Distributed Event-based Systems*. 63–74.

[11] Ashwin R. Bharambe, Sanjay Rao, and Srinivasan Seshan. 2002. Mercury: A Scalable Publish-subscribe System for Internet Games. In *Proc. 1st Workshop on Network and System Support for Games* (Braunschweig, Germany). 3–9.

[12] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proc. 19 Annual ACM Symp. on Principles of Distributed Computing (PODC'00)*.

[13] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *Proc. USENIX Annual Technical Conf. (USENIX-ATC'13)*. 49–60.

[14] C. Cañas, K. Zhang, B. Kemme, J. Kienzle, and H.A. Jacobsen. 2014. Publish/subscribe network designs for multiplayer games. In *Proc. 15th Intl. Middleware Conf.* 241–252.

[15] Lisi Chen, Gao Cong, Xin Cao, and Kian-Lee Tan. 2015. Temporal spatial-keyword top-k publish/subscribe. In *Proc. IEEE 31st Intl. Conf. on Data Engineering*. 255–266.

[16] Paolo Costa, Cecilia Mascolo, Mirco Musolesi, and Gian Pietro Picco. 2008. Socially-aware routing for publish-subscribe in delay-tolerant mobile ad hoc networks. *IEEE Journal on Selected Areas in Communications* 26, 5 (2008), 748–760.

[17] Emitter. 2018. *Scalable and real-time networking: Simple and efficient messaging platform for all of your gaming, IoT and Web apps.* Retrieved Jul 2019 from https://emitter.io

[18] Christian Esposito, Massimo Ficco, Francesco Palmieri, and Aniello Castiglione. 2015. A knowledge-based platform for big data analytics based on publish/subscribe services and stream processing. *Knowledge-Based Systems* 79 (2015), 3–17.

[19] Patrick Th. Eugster, Pascal A. Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. 2003. The many faces of publish/subscribe. *Comput. Surveys* 35, 2 (June 2003), 114–131.

[20] Françoise Fabret, H. Arno Jacobsen, François Llirbat, João Pereira, Kenneth A. Ross, and Dennis Shasha. 2001. Filtering algorithms and implementation for very fast publish/subscribe systems. *ACM Sigmod Record* 30, 2 (2001), 115–126.

[21] Facebook. 2020. *Distributed storage for sequential data.* Retrieved Oct 2020 from https://logdevice.io [See also https://github.com/facebookincubator/LogDevice].

[22] Fastly. 2021. *Compute@Edge.* Retrieved May 2021 from https://www.fastly.com/products/edge-compute/serverless/

[23] Google Cloud. 2021. *Pub/Sub.* Retrieved Aug 2021 from https://cloud.google.com/pubsub/

[24] GraphQL.org. 2021. *A query language for your API.* Retrieved Sept 2021 from https://graphql.org

[25] Abhishek Gupta, Ozgur D. Sahin, Divyakant Agrawal, and Amr El Abbadi. 2004. Meghdoot: Content-based publish/subscribe over P2P networks. In *Proc. ACM/IFIP/USENIX Intl. Conf. on Distributed Systems Platforms and Open Distributed Processing*. 254–273.

[26] A. Hakiri, P. Berthou, A. Gokhale, and S. Abdellatif. 2015. Publish/subscribe-enabled software defined networking for efficient and scalable IoT communications. *IEEE Communications Magazine* 53, 9 (2015), 48–54.

[27] Gaston C Hillar. 2017. *MQTT Essentials-A lightweight IoT protocol.* Packt Publishing Ltd.

[28] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-free coordination for Internet-scale systems. In *Proc. USENIX Annual Technical Conf. (USENIX-ATC'10)*. 145–158.

[29] Cloudfare Inc. 2021. *Cloudfare workers.* Retrieved May 2021 from https://workers.cloudflare.com/

[30] Petri Jokela, András Zahemszky, Christian Esteve Rothenberg, Somaya Arianfar, and Pekka Nikander. 2009. LIPSIN: Line speed publish/subscribe inter-networking. *ACM SIGCOMM Computer Communication Review* 39, 4 (2009), 195–206.

[31] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: A distributed messaging system for log processing. In *Proc. NetDB*, Vol. 11. 31–37.

[32] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. 2017. The Quic transport protocol: Design and internet-scale deployment. In *Proc. Conf. of the ACM special interest group on data communication*. 183–196.

[33] Jon Lee and Wesley Wu. 2019. *How LinkedIn customizes Apache Kafka for 7 trillion messages per day.* Retrieved Apr 2020 from https://engineering.linkedin.com/blog/2019/apache-kafka-trillion-messages

[34] Microsoft Azure. 2020. *Event Hubs.* Retrieved Apr 2020 from https://azure.microsoft.com/en-us/services/event-hubs/

[35] Microsoft Azure. 2021. *Azure Service Bus documentation.* Retrieved Aug 2021 from https://docs.microsoft.com/en-us/azure/service-bus/

[36] Gero Mühl. 2002. *Large-scale content-based publish-subscribe systems.* Ph. D. Dissertation. Technische Universitä Darmstadt.

[37] Emanuel Onica, Pascal Felber, Hugues Mercier, and Etienne RiviÃĺre. 2016. Confidentiality-preserving publish/subscribe: A survey. *Comput. Surveys* 49, 2 (2016), 27:1–27:43.

[38] Lukasz Opyrchal, Mark Astley, Joshua Auerbach, Guruduth Banavar, Robert Strom, and Daniel Sturman. 2000. Exploiting IP multicast in content-based publish-subscribe systems. In *Proc. IFIP/ACM Intl. Conf. on Distributed Systems Platforms*. 185–207.

[39] Victoria Pimentel and Bradford G Nickerson. 2012. Communicating and displaying real-time data with websocket. *IEEE Internet Computing* 16, 4 (2012), 45–53.

[40] Pivotal. 2021. *RabbitMQ is the most widely deployed open source message broker.* Retrieved Aug 2021 from https://www.rabbitmq.com

[41] PubNub. 2021. *The real-time communication platform: Build, integrate, and operate event-driven applications.* Retrieved Aug 2021 from https:

//www.pubnub.com/products/pubnub-platform/

[42] PubNub Inc. 2020. *Innovate with realtime features.* Retrieved Apr 2020 from https://www.pubnub.com

[43] Pusher Ltd. 2020. *Powering realtime experiences for mobile and Web.* Retrieved Apr 2020 from https://pusher.com

[44] Laurin Quast and Stephen Wicklund. 2021. *GraphQL: Real-time with any schema or transport.* Retrieved Sept 2021 from https://github.com/n1ru4l/graphql-live-query/blob/main/README.md

[45] Chuck Rossi, Elisa Shibley, Shi Su, Kent Beck, Tony Savor, and Michael Stumm. 2016. Continuous mobile deployment at Facebook. In *Proc. ACM SIGSOFT Intl. Symp. on the Foundations of Software Engineering (FSE'16).* 12–23.

[46] RSocket.io. 2021. *RSocket: Application protocol providing Reactive Streams semantics.* Retrieved Sept 2021 from http://rsocket.io

[47] Tony Savor, Mitchell Douglas, Michael Gentili, Laurie Williams, Kent Beck, and Michael Stumm. 2016. Continuous deployment at Facebook and OANDA. In *Proc. 38th Intl. Conf. on Software Engineering Companion (ICSE-C'16).* 21–30.

[48] Dan Schafer and Laney Kuenzel. 2015. *Subscriptions in GraphQL and Relay.* Retrieved Sept 2021 from https://graphql.org/blog/subscriptions-in-graphql-and-relay/

[49] Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, and Thomas Fersch. 2015. Wormhole: Reliable pub-sub to support geo-replicated Internet services. In *12th USENIX Symp. on Networked Systems Design and Implementation (NSDI'15).* 351–366.

[50] Gokul R. Subramanian. 2000. *KIP-578: Add configuration to limit number of partitions.* Retrieved Aug 2020 from https://cwiki.apache.org/confluence/display/KAFKA/KIP-578%3A+Add+configuration+to+limit+number+of+partitions

[51] Sasu Tarkoma. 2012. *Publish/subscribe systems: Design and principles.* John Wiley & Sons.

[52] TIBCO Software Inc. 2021. *TIBCO Rendezvous.* Retrieved Aug 2021 from https://www.tibco.com/resources/datasheet/tibco-rendezvous

[53] Victor Vasiliev. 2021. *The WebTransport Protocol Framework.* Internet-Draft draft-ietf-webtrans-overview-02. Internet Engineering Task Force. https://datatracker.ietf.org/doc/html/draft-ietf-webtrans-overview-02 Work in Progress.

[54] WhatWG.org. 2021. *Fetch – Living Standard.* Retrieved Sept 2021 from https://fetch.spec.whatwg.org

[55] WhatWG.org. 2021. *XMLHttpRequest – Living Standard.* Retrieved Sept 2021 from https://xhr.spec.whatwg.org

[56] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. 2015. Hey, you have given me too many knobs!: Understanding and dealing with over-designed configuration in system software. In *Proc. 10th Joint Meeting on Foundations of Software Engineering (FSE'15).* 307âĂŞ319.

[57] ZeroMQ. 2021. *An open-source universal messaging library.* Retrieved Aug 2021 from http://zeromq.org