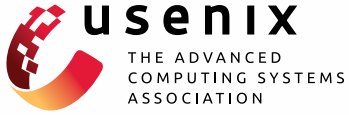

Wormhole: Reliable Pub-Sub to Support Geo-replicated Internet Services

Yogeshwer Sharma, Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary, Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski, Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev, and Kowshik Prakasam, *Facebook;* Robbert van Renesse, *Cornell University;* Sabyasachi Roy, Pratyush Seth, Yee Jiun Song, Benjamin Wester, Kaushik Veeraraghavan, and Peter Xie, *Facebook*

https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/sharma

This paper is included in the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15).

May 4–6, 2015 • Oakland, CA, USA

ISBN 978-1-931971-218

Open Access to the Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15) is sponsored by USENIX

Wormhole: Reliable Pub-Sub to support Geo-replicated Internet Services

Yogeshwer Sharma[†], Philippe Ajoux, Petchean Ang, David Callies, Abhishek Choudhary,
Laurent Demailly, Thomas Fersch, Liat Atsmon Guz, Andrzej Kotulski,
Sachin Kulkarni, Sanjeev Kumar, Harry Li, Jun Li, Evgeniy Makeev,
Kowshik Prakasam, Robbert van Renesse[*], Sabyasachi Roy, Pratyush Seth,
Yee Jiun Song, Kaushik Veeraraghavan, Benjamin Wester, Peter Xie.
Facebook Inc., [*]Cornell University
[†]Contact: yogi@fb.com, sharma@cornell.edu

(Operational Systems Track)

Abstract

Wormhole is a publish-subscribe (pub-sub) system developed for use within Facebook's geographically replicated datacenters. It is used to reliably replicate changes among several Facebook services including TAO, Graph Search and Memcache. This paper describes the design and implementation of Wormhole as well as the operational challenges of scaling the system to support the multiple data storage systems deployed at Facebook. Our production deployment of Wormhole transfers over 35 GBytes/sec in steady state (50 millions messages/sec or 5 trillion messages/day) across all deployments with bursts up to 200 GBytes/sec during failure recovery. We demonstrate that Wormhole publishes updates with low latency to subscribers that can fail or consume updates at varying rates, without compromising efficiency.

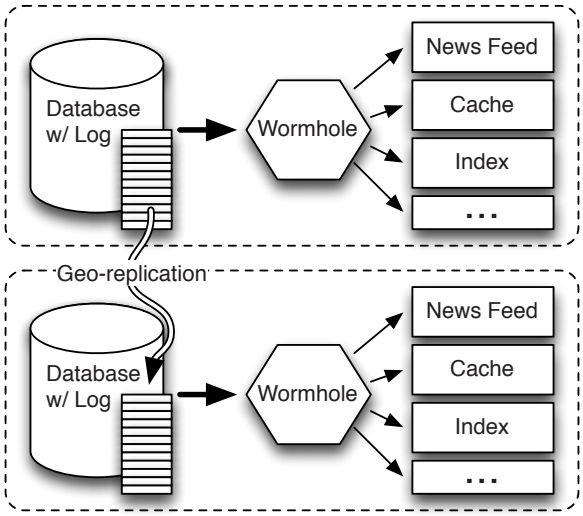

Figure 1: Wormhole reads updates from the data storage system transaction logs and transmits them to interested applications that include News Feed, index servers, Graph Search and many others.

1 Introduction

Facebook is a social networking service that connects people across the globe and enables them to share information with each other. When a user posts content to Facebook, it is written to a database. There are a number of applications that are interested in learning of a write immediately after the write is committed. For instance, News Feed is interested in the write so it can serve new stories to the user's friends. Similarly, users receiving a notification might wish to immediately view the content. A number of internal services, such as our asynchronous cache invalidation pipeline, index server pipelines, etc. are also interested in the write.

Directing each application to poll the database for newly written data is untenable as applications have to decide between either long poll intervals which lead to stale data or frequent polling which interferes with the production workload of the storage system.

Publish-subscribe (pub-sub) systems that identify updates and transmit notifications to interested applications offer a more scalable solution. Pub-sub systems are well studied (see Section 6) with many commercial and open source solutions. However, most existing pub-sub systems require a custom data store that is interposed on writes to generate the notifications for interested applications. This is impractical for Facebook which stores user data on a fleet of sharded storage systems including MySQL databases [18], HDFS [27] and RocksDB [30] across multiple datacenters. Interposing on writes to these storage systems would require modifications across the software stack, which is error-prone and might degrade latency and availability. Writing the updates to a custom data store would also introduce an additional intermediary storage system that might fail.

To address our requirements, we built and deployed

Wormhole, a pub-sub system that identifies new writes and publishes updates to all interested applications (see Figure 1). Wormhole *publishers* directly read the transaction logs maintained by the data storage systems to learn of new writes committed by *producers*. Upon identifying a new write, Wormhole encapsulates the data and any corresponding metadata in a custom *Wormhole update* that is delivered to subscribers. Wormhole updates always encode data as a set of key-value pairs irrespective of the underlying storage system, so interested applications do not have to worry about where the underlying data lives.

Data storage systems are typically geo-replicated with a single master, multiple slaves topology. Wormhole delivers updates to geo-replicated subscribers by piggybacking on publishers running on slave replicas that are close to subscribers. On a write, data is written to the transaction log of the master replica and then replicated asynchronously to the slaves. Wormhole publishers running on the slave can simply read new updates off the local transaction log and provide updates to local subscribers.

Wormhole can survive both publisher and subscriber failures, without losing updates. On the publisher side, Wormhole provides *multiple-copy reliable delivery* where it allows applications to configure a primary source and many secondary sources they can receive updates from. If the primary publisher (which generally is the publisher in the local region) fails, Wormhole can seamlessly start sending updates from one of the secondary publishers. On the subscriber side, an application that has registered for updates can also fail. Wormhole publishers periodically store for each registered application the position in the transaction logs of the most recent update it has received and acknowledged. On an application failure, Wormhole finds where to start sending updates from based on its bookkeeping, maps that to the correct update in the datastore's transaction log and resumes delivering updates.

Wormhole has been in production at Facebook for over three years, delivering over 35 GBytes/sec continuously (over 50 million messages/sec) across all deployments.

Our contributions are as follows:

- We describe the first large scale pub-sub system that delivers trillions of updates per day to subscribers.
- We present a pub-sub system that can run atop existing datastores and provide updates to subscribers.
- We implement *multiple-copy reliable delivery* in Wormhole that allows it to send updates to applications even in the presence of publisher and subscriber failures.
- We allow the datastores to trade-off latency of delivering updates with I/O bandwidth by selecting how much of the disk bandwidth is available for use by Wormhole.

2 Problem

Facebook stores a large amount of user generated data, such as status updates, comments, likes, shares, etc. This data is written to a number of different storage systems depending on several factors such as whether the workload is write optimized or read optimized, what is the capacity versus cost trade-off etc. Moreover, to scale with Facebook's vast user base, these storage systems are sharded and geo-replicated in various data centers.

There are numerous systems that need the newly updated data to function correctly. For instance, Facebook aggressively employs caching systems such as Memcache [19] and TAO [7] so the underlying storage systems are not inundated with read queries. Similarly, Graph Search [12] maintains a index over all user generated data so it can quickly retrieve queried data. On a write, cached and indexed copies of the data need to either be invalidated or updated.

Directing applications to poll the database for newly written data is unscalable. Additionally, writes might be written to any of the different storage systems and applications might be interested in all new updates. Thus, there are a number of challenges that an update dissemination system deployed at Facebook needs to handle:

1. **Different consumption speeds:** Applications consume updates at different speeds. A slow application that synchronously processes updates should not hold up data delivery to a fast one.
2. **At least once delivery:** All updates are delivered at least once. This ensures that applications can trust that they have received all updates that they are interested in.
3. **In-order delivery of new updates:** When an update is received, the application should be confident that all updates prior to the received one have also been received earlier.
4. **Fault tolerance:** The system must be resilient to frequent hardware and software failure both on the datastore as well as the application end.

Challenges 1 and 4 imposed by heterogeneous nature of Facebook's infrastructure, while the others are design choices made based on the nature of applications supported.

3 Wormhole Architecture

In this section, we describe the high level design of Wormhole. Figure 2 shows its main components. In the

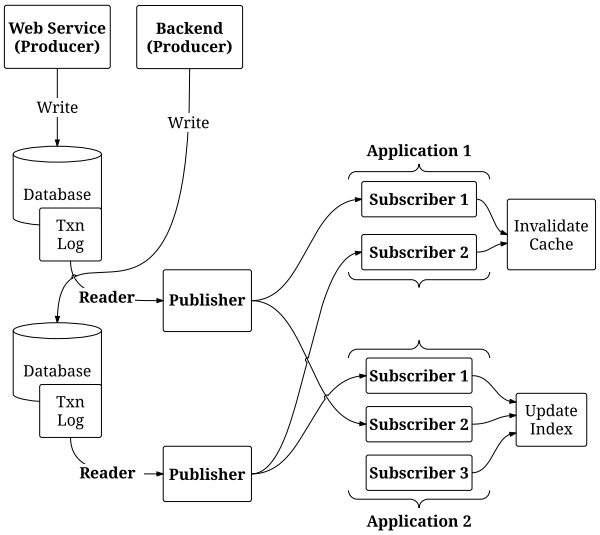

Figure 2: Components of Wormhole. Producers produce data and write to datastores. Publishers read the transaction logs of datastores, construct updates from them, and send them to subscribers of various applications, which in turn do application specific work, e.g., invalidate caches or update indices.

following subsections, we detail specific design choices, their implications, and their role in satisfying the requirements from Section 2.

Data Model and System Architecture. A ***dataset*** is a collection of related data, for example, user generated data in Facebook. It is partitioned into a number of ***shards*** for better scaling, and each update to the dataset is assigned a unique shard. A ***datastore*** stores data for a collection of ***shards*** of the dataset. Each datastore runs a ***publisher***, which reads updates from the datastore transaction log, filters the updates, and sends them to a set of ***subscribers***. Publishers are typically co-located on the database machines so they have fast local access to the transaction log.

In Wormhole publisher, we have support for reading from MySQL, HDFS, and RocksDB, and it is easy to add support for new log types. The ***producers*** of updates write a set of key-value pairs in the log entries in a serialized format that is different for each log type. The publisher takes care of translating the underlying log entries into objects with a standard key-value format, which we refer to as a ***Wormhole update***. One of the keys in the Wormhole updates (called *#.S*) corresponds to the shard of the update: it is written by the producer based on what shard the update belongs to.

Wormhole's subscribing applications are also sharded. An application links itself with the ***subscriber library*** and arranges for multiple instances of itself to be run, called the ***subscribers***. The publishers finds all interested applications and corresponding subscribers via a ZooKeeper-based configuration system. It then divides all the shards to be sent to the application among all of the application's subscribers, eventually calling *onShardNotice()* for notifying subscribers about shards they will be responsible for (see Table 1). It is possible (and likely) that shards belonging to the same publisher might be processed by different subscribers, and shards belonging to different publishers might be processed by same subscriber.

All updates of a single shard are always sent to one subscriber, i.e., they are not split among multiple subscribers. Wormhole arranges for in-order delivery of the updates belonging to any fixed shard: if a subscriber receives an update (say u_1) for a shard (say s_1), then all updates for shard s_1 contained in the transaction logs prior to u_1 must have already been received by the subscriber. Subscribers receive the stream of updates for every shard, which we call a ***flow***. Publishers periodically track ***datamarkers*** per flow after the subscribers acknowledge that they have processed the updates up to new datamarker. A datamarker for a flow is essentially a pointer in the datastore log that indicates the position of the last received and acknowledged update of the flow by the subscriber. Subscribers are assumed to be stateless. In particular, they don't need to keep track of the state of the flow.

Updates Delivery. To get started, a publisher finds applications that want to subscribe to it using configuration files. It constructs flows for these applications corresponding to shards it has in its datastore, constructing one flow for each (application, shard) pair. The configuration can be changed dynamically, for example adding a new application or deleting an old application. This may result in addition or deletion of flows. When a new application is added, it is typically specified which point in the past it wants to start getting updates from. In such case, publishers ensures that it sends updates starting from asked for position.

In steady state, all flows get updates from the same position in the datastore, i.e., the position corresponding to the current time. Hence, Wormhole uses one reader to get the updates, and sends them to all interested flows. In case of error conditions, the publisher needs to restart sending updates from a stored datamarker. For this, the publisher may need to read older updates from the datastore's log. If many flows are recovering simultaneously, a naive implementation of the publisher would read from many positions in the log simultaneously, causing high I/O load. Wormhole clusters flows and creates a reader for each cluster instead, which results in significant I/O savings. Each such reader combined with associated

flows is called a ***caravan*** (see Section 3.1). The singleton steady state caravan (or the farthest caravan in case of multiple caravans) is called the ***lead caravan***.

Wormhole needs to load balance flows among the subscribers of an application. We use two modes of load balancing. In the first mode, the publishers use a weighted random selection to choose which subscriber to associate a flow with, so that lightly loaded subscribers are more likely to get more flows. In the second mode, the subscribers use ZooKeeper [13], a distributed synchronization service, to balance load among themselves: If some subscribers get heavily loaded, they redirect some flows to lightly loaded subscribers. Subscribers can also implement a custom load-balancing strategy using these primitives.

After a flow is associated with a caravan and a subscriber, updates are sent over a TCP connection. Publishers save on connection overhead by multiplexing all flows associated with the same subscriber. The datamarkers are moved forward for flows periodically after corresponding subscribers confirm receipt of new updates.

Wormhole provides libraries to make it easy to build new applications. The subscriber library takes care of communication with the publisher, responding to datamarkers, and other protocol details. The subscribers of the application implement an API that is specified in Table 1.

Before moving on to an in-depth description of Wormhole components, we make a few observations. (1) Wormhole is highly configurable, and therefore can be used for diverse applications such as cache invalidations, index updates, replication, data loading to Hive etc. It has pluggable datastore support, allows configurations to be changed on the fly, and has a configurable flow-clustering algorithm (see Section 3.1). (2) Wormhole supports applications that are written in various languages. Currently applications written in C++ and Java are supported. (3) Wormhole is highly reliable and can handle failures of various components: publishers, datastores, subscribers, and even partial network failures.

3.1 Caravan Allocation

As described earlier, Wormhole publisher has the singleton lead caravan in steady state which reads updates from the datastore and sends them to appropriate subscribers. When some subscribers become slow in processing updates, Wormhole creates additional caravans to follow the lead caravan. These additional caravans send past updates to slow subscribers. In addition, flows are dynamically assigned to a varying number of caravans as their datamarkers change in order to optimize for latency and I/O-load. The trade-off between I/O-load and latency can be intuitively seen as follows: If we allow a large number of caravans, we incur higher I/O-load, but we can do a better job of clustering flows whose datamarkers are close, which prevents other flows from having to wait. If we allow very few caravans, flows with very different datamarkers get assigned to the same caravan, making flows which are farther ahead wait for flows that are very far behind. The allocation of flows to caravans is called ***caravan allocation***. Note that the datamarkers of all flows assigned to a caravan must be at least as large as the position of the caravan in the transaction log.

Caravans are periodically split and merged based on the datamarkers of the flows. A caravan is split if the flows on it can be tightly clustered into more than one cluster. Two caravans are merged if they are "close" to each other and reading updates that are nearby in the datastore transaction log. Usually, the non-lead caravans are expected to eventually catch up with the lead caravan and are thus forced to read updates at a rate that is faster than the rate of the lead caravan (typically 1.25 to 2 times faster). In order to prevent overloading the datastore, Wormhole has configuration parameters for the maximum number of caravans, the maximum rate at which a caravan is allowed to read updates, and a maximum cumulative rate at which the collection of caravans is allowed to read updates.

We also dynamically move flows between existing caravans. If a caravan has a flow which is not able to keep up with the speed of the caravan (because the corresponding subscriber is overloaded, for instance) or whose datamarker is far ahead (and can better served by another caravan), we can move the flow. These actions are taken periodically.

3.2 Filtering Mechanism

Wormhole implements *publisher-side filtering*: the application informs publishers of what filters it needs; the publisher only delivers updates that pass the supplied filters. While evaluation of filters places some additional processing overhead on a publisher, it helps conserve both memory and network bandwidth. The efficiency resulting from publisher-side filtering is more pronounced when there are many applications that need only a subset of data.

Filtering is based on the Wormhole update format, which is a set of key-value pairs. Filters are specified as follows: the top-level filter is an "OR" which is a disjunction of finitely many mid-level "AND," each of which in turn is a conjunction of finitely many "basic filters." A basic filter on an update is one of four kinds: (1) Does a key exist, (2) Is the value of a key equal to a specified value, (3) Is the value of a key contained in a specified set (a numeric interval or a regular expression or a list of

Callback for application	**When is the callback invoked by subscriber library**
onShardNotice(shard)	when updates for a new shard are discovered by the publisher, it notifies the subscriber of existence of a new shard
onUpdate(wormholeUpdate)	when a new update is received from the publisher
onToken(datamarker)	when publisher requests acknowledgement that the subscriber has received data up to the new datamarker
onDataLoss(fromMarker, toMarker)	when the publisher realizes that some data for the flow was not sent

Table 1: API that Wormhole subscribers implement to get updates from publishers. The callbacks specified are run when an event of a specific type happens. *onShardNotice()* is called when a subscriber is notified of a new flow corresponding to a new shard. Once the flow is established, *onUpdate()* is called for each received update. When the publisher sends a new datamarker asking for acknowledgement of received data, *onToken()* is called. When the subscriber has processed all updates up to the supplied datamarker, it is passed back to the publisher as an acknowledgement. In the rare event that truncation of logs by the underlying datastore results in data loss for an application because the application was further behind than the truncated log position, *onDataLoss()* callback is called to notify the subscriber of data loss event.

elements), or (4) negation of any of the previous three.

This generic filtering system is flexible enough for Facebook's various applications. For example, for the cache invalidation applications we use the filter *[topic = aq] OR [mcd key exists]* (*mcd* specifies keys to invalidate in Memcache). For index update services, we use more complex filters such as *[tableName in (t1, t2)] OR [(associationType = a1) AND (shard in 1-5000)]*.

3.3 Reliable Delivery

Reliability is an important requirement for Wormhole. For example, one missed update could lead to permanent corruption in a cache or index of a dataset. Wormhole supports two types of datasets: *single-copy datasets* and *multiple-copy datasets*. The latter indicates a geo-replicated dataset. Accordingly, Wormhole supports both ***single-copy reliable delivery*** (***SCRD***) and ***multiple-copy reliable delivery*** (***MCRD***). For SCRD, Wormhole guarantees that when an application is subscribed to the single copy of a dataset, its subscribers receive *at least once* all updates contained in that single copy of the dataset. The updates for any shard are delivered to the application *in order* that they were stored in the transaction logs: delivery of an update means all prior updates for that shard have already been delivered. For MCRD, applications are allowed to subscribe to multiple copies of a dataset at once, and when they do so, Wormhole guarantees that its subscribers receive *at least once* all updates contained in any subscribed copy of the dataset. The updates for any shard are, again, delivered *in order*. There is no ordering guarantee between updates that belong to different shards. Ordering guarantee for updates within shards suffices for most purposes, since updates corresponding to one entity (e.g., a Facebook page, or a Facebook user) reside on the same shard.

Note that these guarantees do not hold if an update is not available in the datastore log at the time application is ready to receive updates (because datastore might have truncated its logs). Typically datastore logs retain updates for 1–2 days, and an application that falls behind by more than that may thus miss updates (and notified by *onDataLoss()* callback, see Table 1). In our experience, when applications do fall behind because of machines or network failures, monitoring alarms become active and remediation is done quickly. Hence, it is rare for the applications to fall behind by more than a few hours.

In rest of this section, we first show how Wormhole uses datamarkers to provide SCRD, and then how it is extended to MCRD.

3.3.1 Single-Copy Reliable Delivery (SCRD)

Wormhole leverages the reliability of TCP: while a subscriber is responsive, TCP ensures reliable delivery. Wormhole does not use application layer acknowledgements for individual updates—we found it resulted in heavy bandwidth usage and lowered throughput. Instead, for every flow, a publisher periodically sends a datamarker (current position in the datastore log) interspersed with updates. The subscriber acknowledges a datamarker once it has processed all updates before the datamarker. The acknowledged datamarkers are stored on the publisher side in persistent storage. Since the publisher can send updates to a flow only if both the datastore and the datamarker are available, it makes sense to store them together.

These stored datamarkers help the publisher achieve SCRD. When a subscriber becomes available after subscriber or network failure, a publisher uses previously acknowledged datamarkers as starting points for sending updates, hence not missing any update.

The decision of sending datamarkers only periodically results in higher throughput in the normal case. But in the case of a recovery, when publisher starts sending updates from previously acknowledged datamarker, some updates may be received more than once. This is not a problem in most cases. Multiple deliveries of cache invalidation updates do not violate correctness, and other applications can easily built logic to remove duplicates. Wormhole also provides an interface to allow subscribers to send datamarkers to publishers in order to reduce the probability of duplicate delivery.

The average size of datamarkers is less than 100 bytes and they are sent only once every 30 seconds. Hence network overhead from datamarkers is small: 0.0006%–0.0013% of the traffic. This overhead can be reduced by increasing the period between datamarkers, but doing so results in a higher overhead when recovering. With 30 seconds period, 0.2 seconds of updates are resent for every 10 minutes of updates on average because of failures and recoveries, resulting in 0.03% network overhead.

We also support a mode where applications can choose to get only real time data (hence not requiring caravans for old updates), or data that is at most a certain time old. While it does not guarantee reliable delivery, this mode is frequently used for developing and testing applications.

3.3.2 Muliple-Copy Reliable Delivery (MCRD)

Most datasets at Facebook are replicated, allowing for higher availability of updates. In MCRD, when a publisher doing single-source reliable delivery to an application fails permanently, e.g., because of hardware failure, we would like a publisher running on replicated datastore to take over and do single-source reliable delivery to the application starting from where the failed datastore stopped sending updates. In essence, MCRD is "SCRD with publisher failover." There are, however, several challenges in extending the SCRD guarantee to MCRD:

(1) The datamarkers for flows are stored in persistent storage by the publisher, which is typically co-located with the datastore host. When the host fails, we lose the datamarker even though the updates might be available elsewhere.

(2) The datamarker for a flow is a pointer into the logs of the datastore. It is usually a filename of the log and byte offset within that file. Unfortunately, datamarkers represented this way are specific to the particular replica and it is not straightforward to find the corresponding position in a different replica. For example, in MySQL, binary log names and offsets are completely different for different replicas.

(3) For simplicity, publishers are independent entities in MCRD case, and they do not communicate to each other. For ease of operations, we would still like a solution that minimized the communication between publishers.

We address these challenges in the following ways. First, MCRD publishers store datamarkers in ZooKeeper, a highly available distributed service.

To overcome the problem of replica-specific datamarkers, we introduce ***logical positions***—a datastore agnostic way to identify updates such that copies of the same update in different replicas have the same logical position. A logical position uniquely identifies an update in a dataset using a (monotonically non-decreasing) ***sequence number*** and an ***index***. The updates are assigned logical positions by the publisher. When the sequence number of consecutive updates are equal, which can happen if the datastore does not natively support sequence numbers and we use timestamps of updates as sequence numbers, they are assigned monotonically increasing indices starting at 1 so they have unique logical positions. Since caravans still need datastore positions to start reading logs, a data structure called ***logical positions mapper***, or simply ***mapper***, maps logical positions from datamarkers to datastore-specific positions.

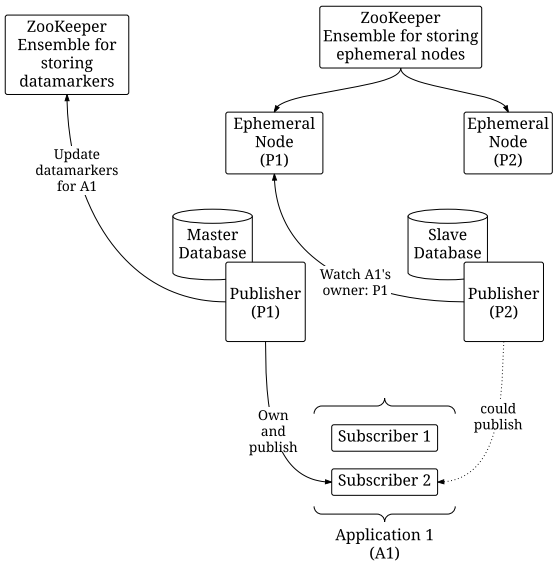

Figure 3: Architecture for the failover of publishers in MCRD. Multiple publishers (in this case P1 and P2) have the same data to publish to application (A1), but only one of them (P1) owns and publishes. Each publisher has an ephemeral node corresponding to it, which non-owner publisher watches in case the owner fails (P2 watching P1). The owner publisher updates datamarkers in ZooKeeper. If P1 fails, P2 will notice the disappearance of P1's ephemeral node and will start owning the flows for the application.

Finally, to reduce the communication among publishers, we note that MCRD publishers need to be aware only of their peers serving the *same* updates. Therefore, a publisher needs to only communicate to as many publishers as there are replicas. At any time, one peer *owns* a flow and other peers *watch* the owner for any changes. The owner publisher is responsible for sending updates and recording logical datamarkers in ZooKeeper. All other peers keep track of the owner (or ZooKeeper's *ephemeral nodes* [13] corresponding to the owner) and take over the ownership if the owner fails. This architecture is illustrated in Figure 3.

4 Workload and Evaluation

Wormhole has been in production at Facebook for over three years. Over this period, Wormhole has grown tremendously from its initial deployment where it published database updates for cache invalidation system. Today, Wormhole is used by tens of datasets stored on MySQL, HDFS and RocksDB. Across these deployments, Wormhole publishers transport over 35 GBytes/sec of updates at steady state across 5 trillion messages per day. Note that this update rate isn't the peak for the system—in practice, we have found that Wormhole has transported over 200 GBytes/sec of updates when some applications fail and need to replay past updates. We have found that designing Wormhole to run alongside sharded datastores has allowed us to scale the system horizontally by bringing up new publishers on the datastore machines.

To keep Wormhole from hurting datastore performance, a typical constraint in production is to not start too many readers to read updates from datastores. As a result, the historic average of number of caravans used by Wormhole publishers in production is just over 1 (≈ 1.063).

Our evaluation of Wormhole focuses on our production deployment and a few synthetic benchmarks that illustrate Wormhole's characteristics. We focus on the following metrics.

Scalability and Throughput: What is Wormhole's publisher and subscriber throughput? How well does it scale with the number of applications subscribing to updates?

Efficiency: Do caravans reduce load on datastores? How well does Wormhole trade-off I/O efficiency with latency in delivering updates?

Latency: What is the typical latency for delivering updates?

Fault Tolerance: How well does Wormhole handle the failure of publishers, subscribers and even a whole datacenter?

4.1 Scalability

4.1.1 Scaling with the number of applications

This experiment evaluates how a single publisher scales with an increasing number of applications.

Methodology. We start one publisher configured to use 4GB of memory and 32 CPUs clocked at 2.6 GHz. The datastore is filled with 5 GBytes of past updates from production traffic with updates having a mean size of 1 KBytes. We run 20 experiments, parameterized by number of applications $n = 1, 2, \ldots, 20$. For each n, we configure n applications to receive updates from the publisher. Each of the n applications have one subscriber, which simply receives all 5 GBytes of updates and increments a counter indicating the number of updates received. The publisher is configured to use only one replay caravan. (The lead caravan is at the end of the logs and not relevant for this experiment.) We measure how long it takes the publisher to send all updates to all n applications (i.e., time to replay), and the rate of sending updates (i.e., throughput).

Results. Figure 4(a) plots the time taken to deliver all updates to all applications. We see the time taken to deliver all queued updates grows linearly with the number of applications. This linear growth stems from the publisher having to schedule each update for delivery to a subscriber of each application.

Figure 4(b) plots the average throughput of the publisher over the time of delivery of all updates. We find that the throughput increases with increasing number of applications before it levels off at just over 350 MBytes/sec. This bottleneck is caused by a lack of parallelism in our publisher, which has not been optimized because Wormhole publishers are typically co-located with production databases and are not allowed to use many cores.

Note that the goal of this experiment is to stress test the publisher by configuring each application to subscribe to all updates. This is in contrast to our production setup where each application typically gets a filtered subset of updates. In our production set up, it is common for publishers to deliver updates to many tens of applications in steady state.

4.1.2 Subscriber throughput

We now turn our attention to the throughput of the subscriber. In this experiment, we stress test a single subscriber by increasing the number of updates the subscriber is configured to get. This is done by increasing the number of publishers whose updates the subscriber is configured to get.

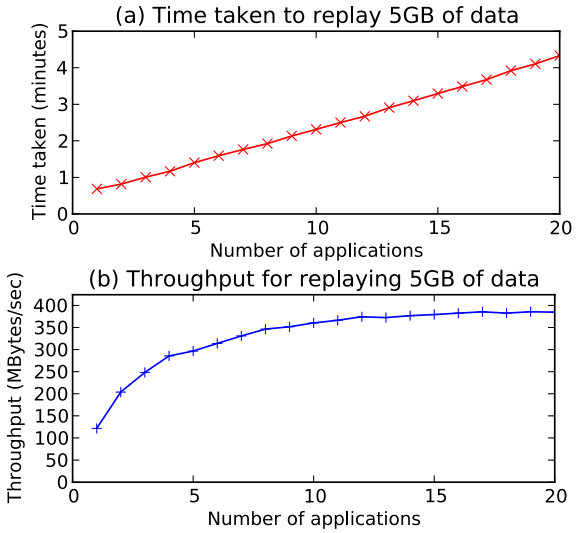

Figure 4: Wormhole publisher delivering updates to varying number of applications from 1 to 20. Panel (a) shows time taken to deliver all updates using a single replay caravan. Panel (b) shows the average throughput of the publisher during delivery of all updates.

Methodology. We start one application using one subscriber configured with 4 GBytes of memory and 4 Intel 2.80 GHz CPUs. We configure this subscriber to get a fraction of updates from a production dataset that averages 500 bytes per update. We periodically increase the fraction of updates the subscriber is configured to get by increasing the number of shards from the dataset that the application is interested in. This is done approximately every 15 minutes. We measure the running average of throughput and latency of the subscriber during the experiment.

The measured data is plotted in Figure 5. Both (a) and (b) show the number of shards whose updates the subscriber is getting over the 150-minute experiment. Note that the step increments in this graph at minutes $\approx 80, 95$, and 110 is by manually changing the configuration of the application to be subscribed to more shards from publishers. Figure 5(a) shows the average throughput of the subscriber for each one minute interval. Figure 5(b) shows the average latency of updates delivery for each one minute interval. Note that this latency is end-to-end—the difference between the time at which update was delivered to the subscriber and time at which it was written to the datastore where publisher is reading. Also note that during the time of low latency, the sum total of send-throughputs of all publishers to the subscriber is equal to the receive-throughput of the subscriber.

Note that the throughput jumps when number of shards jump, which is expected because more publishers start sending updates to the subscriber. Despite this increased throughput, the average latency of updates remains constant at 150ms. But the final jump in throughput around minute 110 is not sustained—the throughput hovers around 600,000 updates/sec. Also, the latency starts increasing without limits at minute 110, since the

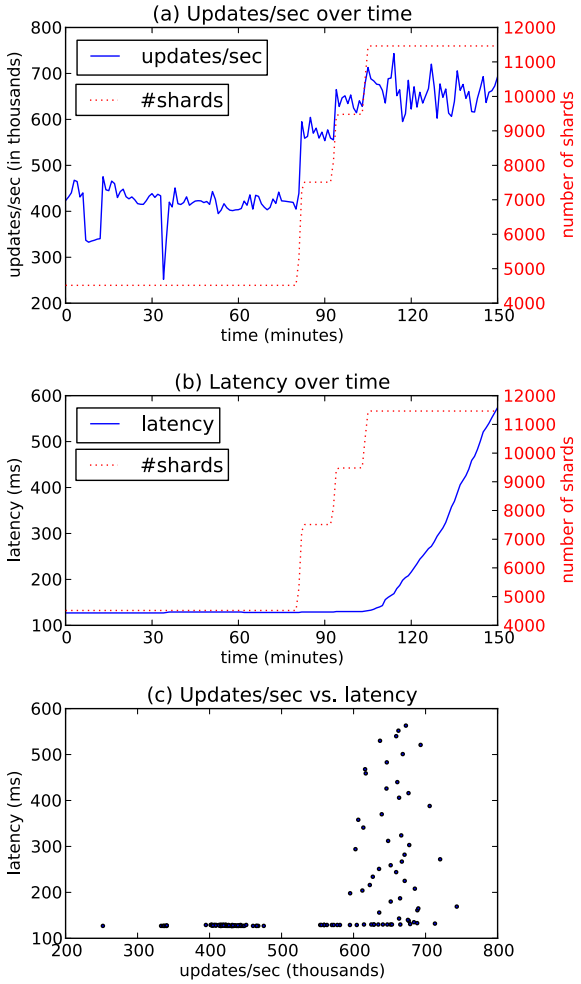

Figure 5: Running average of throughput and latency of subscriber that gets increasingly larger amount of updates from publishers. Increasing amount of updates is shown in (a) and (b) by step increase in number of shards whose updates the subscriber gets. This increase is done manually at minutes $\approx 80, 95$, and 110. Panel (a) shows the throughput in updates/sec, which jumps with jump in number of shards. Panel (b) shows the average latency of delivery of updates, which remains constant up to the throughput limit of the subscriber. Panel (c) combines (a) and (b) by plotting throughput versus latency of each one minute interval shown in (a) and (b). It shows that the latency of updates delivery remains low up to a limit, and increases in an unbounded manner after that.

subscriber is not able to handle all the updates. This leads to publisher having to allocate a different caravan for sending updates again. This time is counted against latency of updates, which results in higher latency for updates that are resent. This lack of ability of subscriber to process more than 600k updates/sec is easily seen in Figure 5(c), which is a convenient combination of the previous two: For each one minute interval, it plots the average throughput versus average latency during that one minute interval. After the throughput hits 600k updates/sec, the latency keeps increasing without any further increase in the throughput.

Results. Wormhole subscriber can sustain a high throughput of over 600,000 updates/second, without imposing a latency penalty. If we try to push more updates than that, the subscriber starts dropping updates, which results in publishers having to resend them and causing large latency.

4.1.3 Publisher throughput in production

A previous experiment showed the throughput of the publisher for replaying updates for data that was not serving production traffic (Figure 4). We now evaluate the typical speed of recovery for application in Facebook's production environment and show that Wormhole is capable to serving high throughput in bursts when applications fall behind.

We consider a production deployment of Wormhole with publishers delivering updates to a cache invalidation application. For this application, any delay in delivering updates results in stale cache data.

Methodology. We evaluate 50 publishers over a 24-hour period and report what is the average throughput of the publishers, and what is the maximum throughput of the publishers observed during the experiment. To count towards the maximum, the throughput had to be sustained for at least one minute. We plot the average throughput versus maximum throughput in Figure 6.

Results. The main take-away from this experiment is that in Facebook's production environment, Wormhole publishers are capable of sustaining throughput that is more than 10 times their average throughput. This result is important since it is common for applications to fall behind. When requested, the publisher must be able to help the application recover quickly by sustaining high throughput for short bursts of time.

Note that in the above production environment, the highest throughout a caravan can achieve was artificially capped in the configuration of these publishers so that Wormhole does not adversely affect the performance of the underlying datastore. In the absence of such constraints, Wormhole is capable of higher throughput but may result in worse datastore performance for other clients.

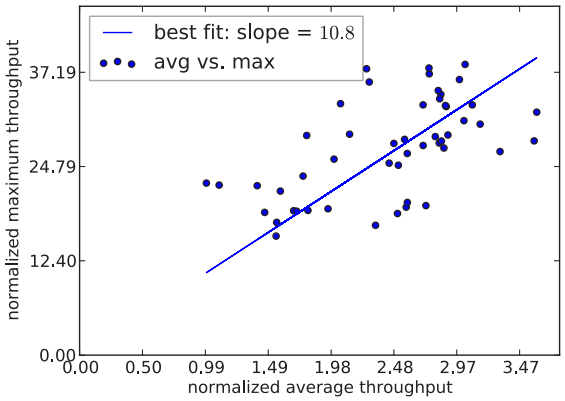

Figure 6: Sustained maximum throughput of publisher versus average throughput over a period of 24 hours for a set of 50 publishers. The throughputs are normalized by making minimum average throughput equal to 1 unit.

4.2 Efficiency

In this section, we evaluate the efficiency achieved by Wormhole by using caravans.

4.2.1 Trading off latency for I/O during recovery

After a failure, multiple applications might fall behind and request updates from different points in time in the past. Wormhole has a choice of creating many caravans starting at different positions in the datastore logs, or using fewer caravans and clustering flows from applications. On one extreme, if multiple caravans are spawned, applications can receive updates immediately, resulting in low latency of update delivery but high read amplification (i.e., updates being read multiple times by different caravans). On the other extreme, if we are allowed to start only one caravan, applications that are further along have to wait for lagging applications to recover before getting updates. This results in higher latency for applications that are up to date. We simulate this scenario in evaluating Wormhole's trade-off between I/O and latency.

Methodology. We start a single publisher on a datastore that has 20 GBytes of updates. To simulate multiple applications that fall behind by different amounts in production, we subscribe this publisher to 10 applications whose datamarkers are equally distributed across the 20 GBytes of updates. Therefore, each application wants

to get a progressively smaller tail-end of the updates in the datastore: first application wants to get the whole 20 GBytes, the second application wants to get the last 18 GBytes, and so on, and tenth application wants to get the last 2 GBytes of updates.

Updates from the publisher to all applications can be sent using different number of caravans (which results in different read amplification factors). We run the experiment in 7 iterations, with different maximum number of caravan Wormhole publisher is allowed to use: 10, 7, 5, 4, 3, 2, and 1. For each iteration, we measure how much data is read collectively by all caravans and divide it by total amount of data in the datastore to get the read amplification factor of the iteration. This is plotted on the *x*-axis in Figure 7.

We measure the latency as follows. The latency of one update is measured as the difference between time of receipt of update by application and time of commit of the update in the datastore. The latency of one application is the average latency over all updates it received. The latency plotted on *y*-axis is the average latency of all 10 applications. Since we are replaying past data spanning several minutes, the latency as measured above is expected to be in minutes.

Each data point in Figure 7 corresponds to one iteration of the experiment, indicating the read amplification factor and the average latency of all applications. The label for the data point corresponds to the maximum number of caravans allowed in that iteration.

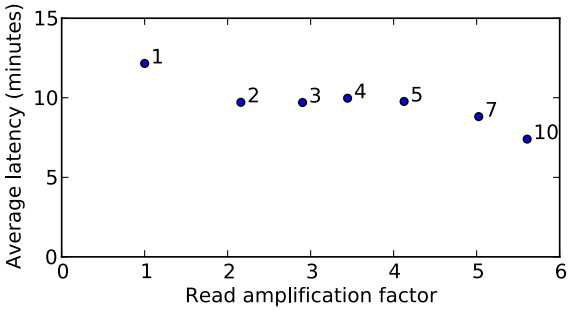

Figure 7: Read amplification factor versus average latency of delivering updates to 10 applications (see text for more explanation). Different data points are for varying number of maximum allowed caravans. The latency is averaged over 10 applications whose datamarkers are scattered evenly across the 20 GBytes of single datastore updates.

Results. The main result of this experiment is to demonstrate that Wormhole can trade off the load on datastore for latency of serving updates. Figure 7 shows that by increasing read amplification on the datastore, Wormhole is able to reduce the average latency of updates by up to 40%.

Note that each addition caravan does not reduce the latency of updates. This is an artifact of how we assign flows to caravans. We believe this can be improved with a different caravan allocation algorithm. This is an active direction for future research, see Section 7.

4.2.2 Updates delivered versus updates read

Methodology. We evaluate how many bytes of updates Wormhole publishers read for each byte of updates sent to all applications. A lower number for this metric indicates Wormhole publisher puts little load on datastores in order to send updates to many applications.

We use measurements from a production deployment that is used to replicate (cache) data across datacenters. There are multiple publishers in the datacenter we are considering, and 6 applications subscribed to the dataset corresponding to these publishers (the number of publishers and subscribers is not relevant to this discussion). The publisher and subscribers are in geographically distributed locations (publisher on the east coast, subscribers on east and west coasts of the US, and Europe). Over a period of 48 hours, we observe the number of bytes collectively sent by these publishers to 6 applications, number of bytes read from datastores by the publishers, and how many caravans were used to read those bytes. These metrics are collected every 1 minute, and the collected value is average of the values since previous collection. The results appear in Figure 8.

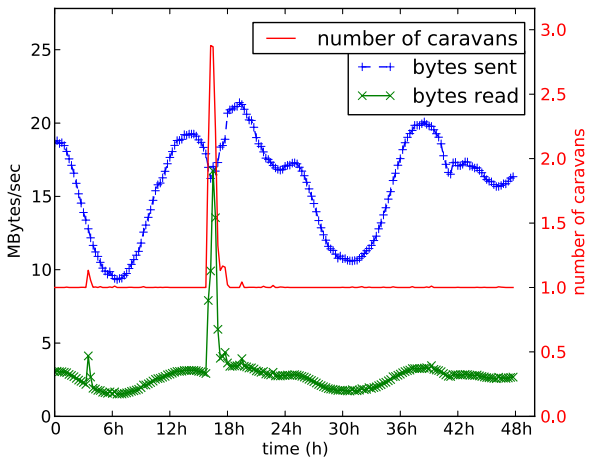

Figure 8: The amount of data read from datastores and sent to applications by Wormhole, and the number of caravans used to do so. The number of caravans, which is averaged over one minute intervals and over all involved publishers, can be fractional.

Results. We see that for a vast majority of time, the ratio of bytes read over bytes sent remains as low at $1/6$. This shows the efficiency of Wormhole in reading updates few times, before sending them to many applications.

There are small spikes in bytes read graph, which corresponds to times when some subscribers of some applications have errors, and hence fall behind. During these spikes, the number of caravans go up accordingly.

Note the large spike in the middle, where the bytes read becomes close to bytes sent (the ratio very close to 1). This happens because many subscribers of one application (out of 6) fall behind because of a systemic error, and all publishers have to read past updates to send past data to the application. Note that even though the number of caravans becomes (only) 3 (for 6 applications), the ratio of bytes read to bytes sent comes close to 1 (instead of 0.5 or $3/6$). The reason is that the two extra non-lead caravans read updates at a higher rate than the lead caravan.

This experiment suggests Wormhole is efficient, even in a large deployment, in reading updates few times and sending them to many applications, and in its ability to recover from applications failures.

4.3 Low Latency

Methodology. We examine the latency experienced by updates in Wormhole. We use the data from one of Facebook's production deployments: we pick one (random) publisher that is sending updates to a cache invalidation application, and observe it over a period of few hours to get a sample of 50,000 updates. The cache consistency application itself runs on hundreds of machines, but the updates from one publisher are distributed to a small number of them (1–3). The average size of updates is 500 bytes. The publisher and subscribers reside in the same datacenter for this setup: the latency between publisher and subscribers (measured via ping time) is low (under 1 ms).

The latency of updates is measured as follows: the publisher writes the (millisecond) timestamp for each update indicating when update was written to the datastore (which could be much earlier than when it was read by the publisher). When the subscriber receives it, it computes the difference of clock time and the timestamp written in the update. The clock skew between the publisher and the subscriber was measured to be less than 1ms. Note that this latency does *not* measure the time taken by the cache consistency application to invalidate the cache.

Results. The cumulative distribution function of latencies of 50,000 updates appear in Figure 9. Over 99.5% of the updates are delivered in under 100ms. Note that there is a long tail of updates that can take as long as 5 seconds. This can happens when an update is sent by a non-lead caravan. In such case, it includes the time that the corresponding flow spends waiting to be assigned to a caravan.

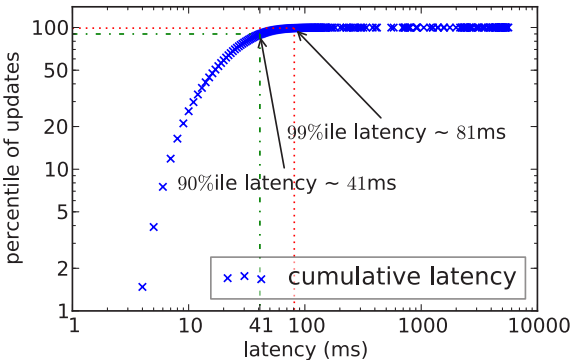

Figure 9: Latency of updates over a period of time, sent from one publisher to local (datacenter) application.

4.4 Reliability and Fault Tolerance

In this section, we evaluate the efficacy of Wormhole in providing multiple-copy (and single-copy) reliable delivery (MCRD and SCRD) by doing a failover for all publishers within a datacenter, by causing single publisher failure, and by causing subscriber failures.

4.4.1 MCRD at large scale

To demonstrate MCRD at large scale, we picked one application that was receiving 300 MBytes/sec of updates from production. We simulated the failure of a datacenter by changing the configuration of the application to get updates from secondary datacenter, instead of the primary datacenter. The ping time between secondary datacenter and application subscribers was 15ms.

Averaged over multiple failovers, it took Wormhole approximately 5 minutes to transfer all traffic from primary to secondary datacenter. A majority of this time is spend during timeouts (for example, it takes one minute before a machine is considered not reachable).

Note that in the case of actual failure of datacenter, the application would not be receiving any data during this time, and receive a burst afterwards when the updates are being sent from the secondary datacenter (qualitatively similar to the bursts in Figure 10(b)).

4.4.2 Reliability under single publisher failure

In this section, we evaluate how Wormhole handles the failures of publishers for SCRD and MCRD.

Methodology. We select two datastores that are replica of a datastore in production. We start one publisher each on them, which serve as peer publisher for MCRD. Call them primary and secondary. We then start two applications: the first one requires multiple-copy reliable delivery of updates, from either primary or secondary, with the preference of primary first. The second application requires single-copy reliable delivery of updates, from the primary publisher only.

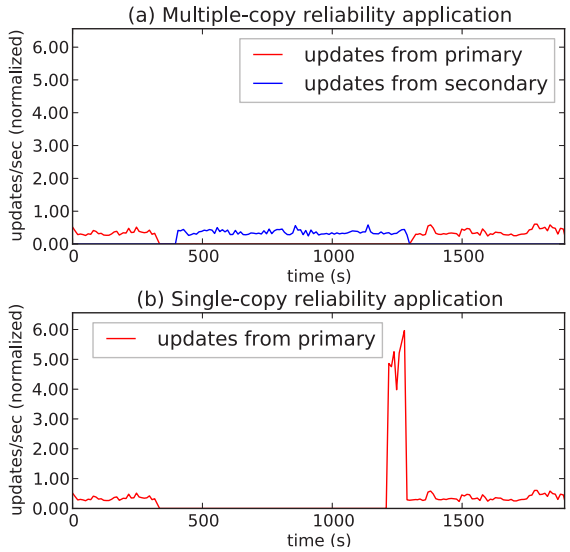

Figure 10: Effect of publisher failure on delivery of updates. Panel (a) shows an MCRD application that is configured to receive updates from either the primary publisher or the secondary publisher. Panel (b) shows an SCRD application that is configured to receive updates only from the primary. The primary publisher is failed close to second 300, and restored at second 1300. In (a), the application starts getting updates from secondary, while in (b), the application has to wait until primary is restored, resulting in higher backlog.

After both applications are receiving data from the primary, we simulate the failure of primary by killing the publisher process on primary (seen by disappearance of red line in Figure 10(a) and (b). We restart the primary publisher 15 minutes after its failure. Note that the secondary publisher runs without any failures.

Figure 10(a) plots the number of updates delivered to the MCRD application from primary and secondary publisher. Figure 10(b) plots the number of updates delivered to the SCRD application from the primary publisher.

Results. This experiment demonstrates the reliable delivery guarantee of Wormhole, showing that both applications survive the failure of primary publisher, albeit in different ways.

The MCRD application starts receiving updates from the secondary publisher within 60 seconds of primary failing. This time comes from the timeout we use to indicate that a publisher is not available any more. When the primary publisher is restored, the MCRD application seamlessly switches to it.

Note the large spike in the updates received for SCRD application. When the only publisher that could deliver updates to it is restored, it sends the backlogs of updates at a higher throughput, and then restores the application to normal state.

4.4.3 Subscriber failures and load balancing

We evaluate Wormhole's capability to balance load among the subscribers of the same application. As described earlier, we use two methods to distribute flows among subscribers: (1) the publisher uses a probability density function to assign flows to subscribers that have relatively fewer flows, and (2) subscribers use a ZooKeeper based load balancing method to provide hint to publisher in choosing subscribers for flows. In the latter method, ZooKeeper based service assigns shards to subscribers, and rebalances periodically when it discovers that the assignment is not balanced. The period is configurable, but typically once per minute.

Methodology. In this experiment, we consider a large number of publishers spanning many shards, say n, delivering updates to a production application that also has a large number of subscribers, say m. We first use one algorithm to balance load among subscribers and then use the second algorithm.

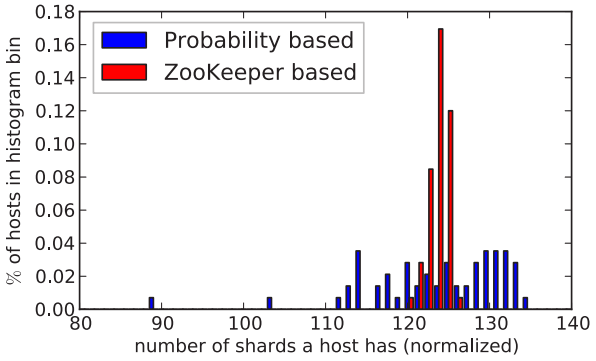

Figure 11: The histograms for distribution of flows among subscribers. The y-axis shows the fraction of subscribers that have number of flows that fall in the horizontal x-axis bin. The two histograms show the spread for probability based distribution and ZooKeeper based distribution.

To measure the efficacy of the load balancing algo-

rithm, we find out how many shards each subscriber is subscribed to. (The sum of these numbers is always n, since there are n shards in total.) We normalize this number and put subscribers in 50 bins according to normalized number of shards assigned to them. This is the x-axis in Figure 11. We draw the histogram in Figure 11 where y-axis shows the fraction of subscribers that have the number of shard subscribed to them on the x-axis. A large spread (as in blue histogram) shows that some subscribers have very few shards while others have very many.

Results. As seen in Figure 11, for the ZooKeeper based allocation policy, the spread of number of shards is tightly concentrated. Most subscribers have between 122 and 126 (normalized) shards. On the other hand, a random distribution assigns as few as 88 and as many as 135 shards to subscribers.

5 Operational Challenges

In this section, we discuss the challenges we have faced in running Wormhole at the scale of Facebook's infrastructure, how we addressed them, and how the system has evolved in a way to make it easy to catch and fix problems.

First, the impact of malfunctioning of Wormhole affects some Facebook users much more than others. For example, suppose Wormhole publishers are malfunctioning on 1% of datastore machines so that 1% of the cache is stale. This would cause 100% of cached data for 1% of the users to be stale—not 1% of the cached data for 100% of the users. This makes the reliability of Wormhole publishers all the more important.

Wormhole must run on all production-ready datastore machines, a large set that keeps changing as machines are brought in to and taken out of production. In our experience with a central deployment system we used early on, it is challenging to keep Wormhole publisher running on exactly this set of machines and no other; a central deployment to a large set of machines is likely to result in some mistakes. We decided to switch to a distributed deployment system based on *runsv* [20] while trying to minimize dependencies outside of the local machine. We run a lightweight ***Wormhole monitor*** on each datastore machine. The monitor periodically checks the configuration system (which indicates the machines that are in production) and based on that determines whether to run a publisher or not, and if so, with what configuration. This decentralized system has been significantly more reliable and easier to use.

In order to make debugging the publisher and fixing problems easier, we can change configurations on-the-fly. For example, the maximum rate at which a caravan can send updates can be changed without restarting the publisher. The publisher also implements a *thrift interface* [4]—we use it to access state (e.g., what are the datamarkers of all flows on it) and gather monitoring statistics that the publisher collects and aggregates every 30 seconds (e.g., the rate of updates sent to all applications). The publisher collects over 100 such monitoring statistics that we use to determine the health of the group of publishers. This interface can also be used to give commands to the publisher to override some decisions manually (e.g., reassign all flows to caravans in order to improve I/O utilization), although this is rarely required.

Wormhole's resource utilization depends not only on its own health, but the health of all the subscribing applications. The difference between best-case resource utilization (one caravan in case all applications are current) and worst-case resource utilization (maximum number of caravans) can be large. We have to plan resources for such worst-case operability, e.g., having a resource limit that is higher than normal operating range for Wormhole.

6 Related Work

The pub-sub systems have been an active area of research for many decades. Many pub-sub systems, message buses (topic-based pub-sub systems), and P2P notification systems have been developed [5,6,9,11,16,17,24,25, 28,31] that shares similar goals to Wormhole. Most solutions, though, use *brokers*—intermediate datastores that store and forward updates. These brokers offload the responsibility of forwarding events to subscribers from the datastores. They can provide reliability by buffering updates for slow subscribers, while providing low latency to fast subscribers. However, these solutions are undesirable for us as they require additional infrastructure for brokers: we do not need brokers to buffer updates as our datastores already provide reliable logs in form of transaction logs. Also, brokers can add significant latency to message delivery, particularly if used hierarchically for large scale systems.

Below we consider some of the best known publicly available products.

SIENA [8] is a wide-area content-based pub-sub service. Much of the focus in SIENA is on its specialized content-based routers, while Wormhole uses stock network routers while filtering happens directly at the publishers. SIENA does not support replicated data sources, and has not been demonstrated at a scale or load near Facebook's.

Thialfi [1] is Google's cloud notification service that addresses a similar problem to Wormhole, namely the invalidation of cached objects. Thialfi is geographically distributed and highly reliable, even in the face of long

disconnections. However, its clients are applications running in browsers on end-users' mobile phones, laptops, and desktops, not applications within Google itself, such as caching and indexing services. The workloads on Thialfi are, accordingly, very different from Wormhole. In particular, Thialfi is not concerned about I/O efficiency on data sources. Thialfi also sends only version number for the data to subscribers, and coalesces many updates into most recent one. Wormhole, on the other hand, sends all updates, and each update contains more information needed by the application, not just the version number. This is for two reasons: (1) By sending data, we can do cache refills, instead of just cache invalidations, and (2) Wormhole is used for wider purposes, such as RocksDB replication, that need each update being delivered (based on statement replication).

Kafka [15] is LinkedIn's message bus, now open source and maintained by Apache. It has a topic-based pub-sub API. Like Wormhole, it uses ZooKeeper to keep track of how many events particularly subscribers have consumed. As in Wormhole, data sources are sharded. At LinkedIn, it is used to distribute various real-time logging information to various subscribers. Kafka can lose messages in case one of its message brokers suffers a failure. Recent benchmarks puts the speed at which Kafka can transport messages at about 250 MBytes/sec, orders of magnitude below Wormhole's production load, but Kafka's throughput can be improved by sharding differently.

Hedwig [3] is a topic-based Apache pub-sub system with an emphasis on handling many topics and providing strong reliability, not on many publishers or subscribers or on high message load. Many pub-sub systems focus on expressive filters and implement sophisticated ways to filter updates [10, 21, 26].

Messages buses like IronMQ [14] and Amazon SQS [2] are hosted, and cannot be installed in local infrastructure. Beanstalkd [22] and RabbitMQ [23] are popular efficient open-source message buses. Beanstalk supports reliability but is specialized to be used as a collection of task queues. Like Wormhole, RabbitMQ can scale to multiple datacenters and is particularly efficient for small messages. Neither supports replicated data sources, or have been been demonstrated to support the scale of Facebook's workloads.

TIBCO Rendezvous [29] is perhaps the most used and advanced commercial message bus. While it has impressive features and performance, it does not support replicated datastores out-of-the-box. Rendezvous also needs additional storage for messages, which grows with the *reliability interval*, during which message can be retransmitted. The Rendezvous daemon does not guarantee delivery to components that fail and does not recover for periods exceeding the reliability interval, which is a different order of magnitude (typically 60 seconds) than the failure durations of Wormhole components (sometimes many hours).

7 Future Work

As Wormhole continues to support the growing amount of traffic flowing through it, there is need for different features to support the load and diversity of use-cases. Because of the growth in number of applications, we are working to provide differentiated guarantees to applications based on how important fresh data and latency is to them. For example, if an application can afford to get data that is stale up to a few minutes, updates for that application can be batched and compressed to save network bandwidth. We are also working on making it easy to swap in and out various caravan allocation policies in the Wormhole publisher, and measure their efficacy for different workloads.

8 Conclusion

This paper describes Wormhole, a pub-sub system developed at Facebook. Wormhole leverages the transaction log of the storage system to provide a reliable, in-order update stream to interested applications. We have demonstrated that Wormhole scales to support multiple data storage systems and can guarantee delivery in the presence of both publisher and subscriber failure. Our production deployment of Wormhole transfers over 35 GBytes/sec in steady state (over 5 trillion messages per day) from geo-replicated datastores to multiple applications with low latency.

Acknowledgements

We would like to thank Nathan Bronson, Daniel Peek and Jonathan Kaldor for reading earlier drafts of this paper and helping improve it. We would also like to thank the users of Wormhole at Facebook, who provided us with the workloads and insights to help design, implement, and scale Wormhole. We are grateful to anonymous NSDI reviewers and our shepherd Atul Adya for their feedback and detailed comments which helped improve the paper.

References

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

[2] Amazon Web Services, Inc. Amazon simple queue service. http://aws.amazon.com/sqs/, 2014.

[3] Apache Software Foundation. HedWig. https://cwiki.apache.org/confluence/display/BOOKKEEPER/HedWig, 2014.

[4] Apache Software Foundation. Thrift. https://thrift.apache.org/, 2014.

[5] R. Baldoni, C. Marchetti, A. Virgillito, and R. Vitenberg. Content-based publish-subscribe over structured overlay networks. In *Distributed Computing Systems, 2005. ICDCS 2005. Proceedings. 25th IEEE International Conference on*, pages 437–446. IEEE, 2005.

[6] R. Baldoni and A. Virgillito. Distributed event routing in publish/subscribe communication systems: a survey (revised version). Technical report, Dipartimento di Informatica e Sistemistica, Università di Roma la Sapienza, 2006.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkataramani. TAO: Facebook's distributed data store for the social graph. In *Proc. 2013 USENIX Annual Technical Conference*, pages 49–60, 2013.

[8] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *Proc. 19th ACM Symposium on Principles of Distributed Computing*, pages 219–227, 2000.

[9] M. Castro, P. Druschel, A.-M. Kermarrec, and A. I. Rowstron. Scribe: A large-scale and decentralized application-level multicast infrastructure. *Selected Areas in Communications, IEEE Journal on*, 20(8):1489–1499, 2002.

[10] A. Demers, J. Gehrke, M. Hong, M. Riedewald, and W. White. Towards expressive publish/subscribe systems. In *Advances in Database Technology-EDBT 2006*, pages 627–644. Springer, 2006.

[11] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, 2003.

[12] Facebook, Inc. Graph search. https://www.facebook.com/about/graphsearch, 2014.

[13] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed. ZooKeeper: Wait-free coordination for internet-scale systems. In *Proc. 2010 USENIX Annual Technical Conference*, 2010.

[14] Iron.io, Inc. IronMQ. http://www.iron.io/mq, 2014.

[15] J. Kreps, N. Narkhede, and J. Rao. Kafka: A distributed messaging system for log processing. In *Proc. 6th ACM Workshop on Networking Meets Databases*, 2011.

[16] Y. Liu and B. Plale. Survey of publish subscribe event systems. Technical Report TR-574, Computer Science Dept., Indiana University, 2003.

[17] S. P. Mahambre, M. K. S.D., and U. Bellur. A taxonomy of QoS-aware, adaptive event-dissemination middleware. *IEEE Internet Computing*, 11(4):35–44, 2007.

[18] MySQL AB. Mysql. http://www.mysql.com/.

[19] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proc. 10th USENIX Symposium on Networked Systems Design and Implementation*, pages 385–398, 2013.

[20] G. Pape. runit. http://smarden.org/runit/, 2014.

[21] M. Petrovic, I. Burcea, and H.-A. Jacobsen. S-ToPSS: Semantic toronto publish/subscribe system. In *Proc. 29th Conference on Very Large Data Bases*, pages 1101–1104, 2003.

[22] Philotic, Inc. Beanstalk. http://kr.github.io/beanstalkd/, 2014.

[23] Pivotal Software, Inc. RabbitMQ. http://www.rabbitmq.com/, 2014.

[24] M. Platania. *Ordering, Timeliness and Reliability for Publish/Subscribe Systems over WAN*. PhD thesis, Sapienza University of Rome, 2011.

[25] V. Ramasubramanian, R. Peterson, and E. G. Sirer. Corona: A high performance publish-subscribe system for the world wide web. In *Proc. 3th USENIX Symposium on Networked Systems Design and Implementation*, pages 15–28, 2006.

[26] W. Rao, L. Chen, A.-C. Fu, H. Chen, and F. Zou. On efficient content matching in distributed pub-/sub systems. In *INFOCOM 2009, IEEE*, pages 756–764. IEEE, 2009.

[27] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The hadoop distributed file system. In *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, MSST '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[28] R. Strom, G. Banavar, T. Chandra, M. Kaplan, K. Miller, B. Mukherjee, D. Sturman, and M. Ward. Gryphon: An Information Flow Based Approach to Message Brokering. In *Proc. 9th Symposium on Software Reliability Engineering*, 1998.

[29] TIBCO Software, Inc. TIBCO rendezvous concepts. `https://docs.tibco.com/pub/rendezvous/8.3.1_january_2011/pdf/tib_rv_concepts.pdf`, 2014.

[30] `https://github.com/facebook/rocksdb`. Rocksdb. `http://rocksdb.org/`.

[31] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowicz. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. 11th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, NOSSDAV '01, pages 11–20, New York, NY, USA, 2001. ACM.