**VIVEKANAND EDUCATION SOCIETY’S INSTITUTE OF TECHNOLOGY**

**Department of Computer Engineering**



Case Study on:

**Twitter as a Distributed System**

As a part of Distributed Computing subject of the Fourth Year, Bachelor of Engineering (B.E.) Degree in Computer Engineering at the University of Mumbai Academic Year 2020-2021.

**Submitted by**

Anish Adnani (D17B - 01)

Nihal Bhandary (D17B - 10)

Supriya Patil (D17B - 50)

**Under the guidance of**

**Dr. Mrs. Anjali Yeole**

**Mrs. Sunita Sahu**

(2020-21)

**TABLE OF CONTENTS**

| **SR. NO** | **TOPIC** | **PAGE** |
| --- | --- | --- |
| 1. | INTRODUCTION | 3 |
| 2. | COMMUNICATION | 6 |
| 3. | TRANSPARENCY | 9 |
| 4. | SYNCHRONIZATION | 10 |
| 5. | CONSISTENCY AND REPLICATION | 12 |
| 6. | FAULT TOLERANCE | 14 |
| 7. | SECURITY | 16 |
| 8. | CONCLUSION | 19 |
| 9. | REFERENCES | 20 |

**INTRODUCTION**

With the rise of the internet, the online community has been growing at an extreme pace due to which internet services have become more complicated.

Twitter is one of the most complicated distributed systems deployed as of now

Below mentioned are some statistics of Twitter

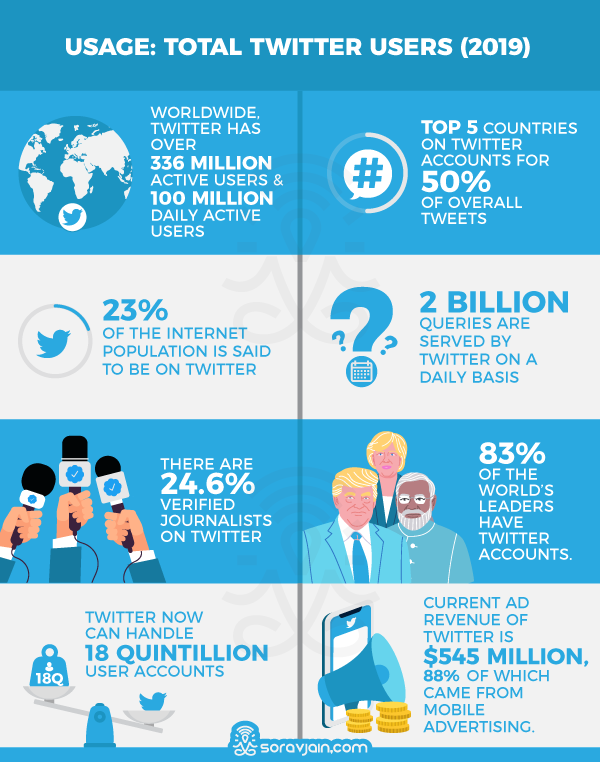
* 400m tweets a day on average
* 5k tweets/sec (tweets per second or “tps”) is the daily average, 7k tps is the daily peak, 12k tps is the peak for some international event and 150k tps is the maximum observed
* 300m read requests per second on average, 50% of them finishes in 3.5 seconds, 99% is up to 5 minutes
* Some of the users have millions of followers
* 336m of active users

The below-mentioned Figure 1 indicates that Twitter is a new era Big Data Distributed System with about 336 Million users, moreover, 100 Million daily active users. It is also stated that 23% of the internet population is said to be on Twitter.

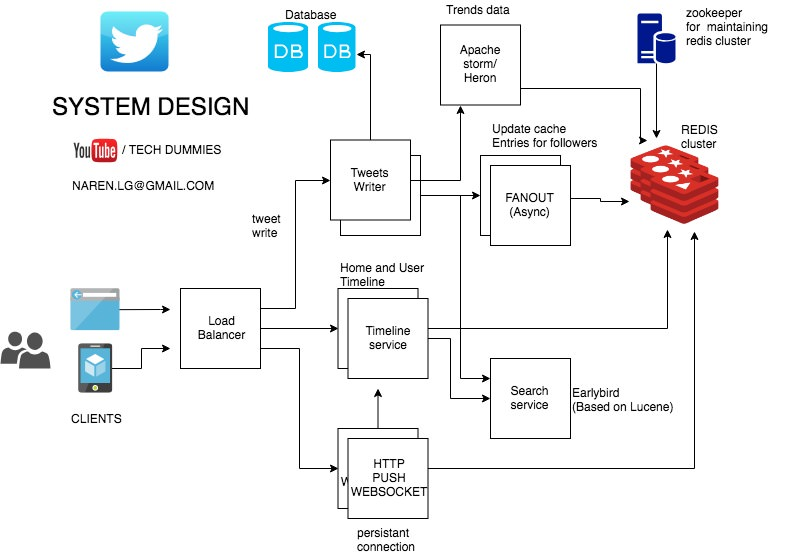
There are about 2 Billion queries that are served by Twitter on a daily basis. Also, Twitter has seen to be one of the most politically persuaded platforms, the very reason which 83% of the world’s political leaders have a Twitter account which they use to make new supporters and engage the targeted audience to a much larger scale than anyone could have ever imagined.

Currently, Twitter is estimated to generate a whooping revenue of about 545 Million Dollars, which is much more than the GDP of many small countries.

With the rise of the internet, one can only imagine Twitter growing its audience.



**Fig 1: Twitter Statistics**



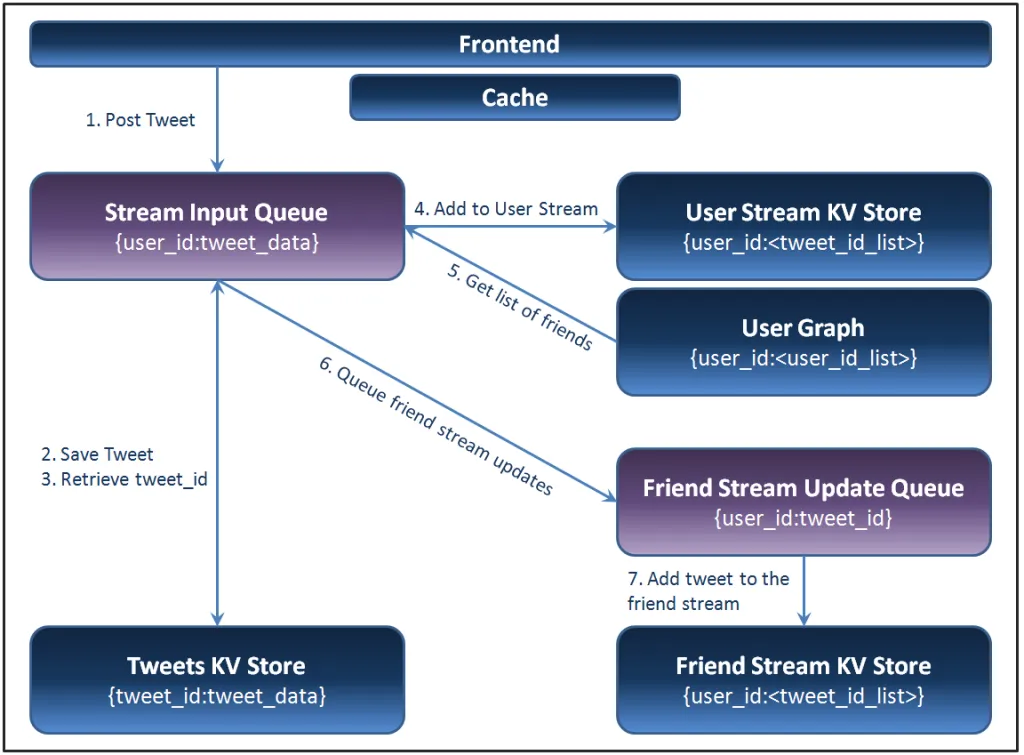
**Fig 2: System Design of Twitter**

**COMMUNICATION**

The microservice architecture comprises many different services with varying capacity requirements. As parts of the Twitter application grow, we can scale demands on capacity by adding more instances or replicas to a respective service cluster (i.e., horizontal scaling). In order to properly utilize all the replicas of a service cluster, a client-side load balancer can be enabled in every client. This allows it to operate with fewer layers of physical infrastructure. In modern serving systems, these load balancers are often referred to as application load balancers and they serve two primary functions:

* They allow callers of a backend service to safely utilize the aggregate capacity by dividing the work among backend replicas.
* Because all requests to a respective service now flow through a load balancer, it is well-positioned to route around replicas when they inevitably fail.

The load balancers balance over both sessions (OSI L5) and requests (OSI L7).



**Fig 3: Explaining the communication**

* There are Caches available for each user, which updates their followers tweets, retweets, etc for faster access rather than searching the whole database
* Twitter uses Redis , Redis basically gives mapping between unique identifier (user\_id) and other notables like their tweets, it contains its own data structure to map list of Tweets to the user\_id
* 
* In order to properly utilize all the replicas of a service cluster, a client-side load balancer can be enabled in every client. This allows it to operate with fewer layers of physical infrastructure

Redis Data structure has the following commands:

***ACL LOAD*** - When Redis is configured to use an ACL file (with the aclfile configuration option), this command will reload the ACLs from the file, replacing all the current ACL rules with the ones defined in the file. The command makes sure to have an *all or nothing* behavior, that is:

* If every line in the file is valid, all the ACLs are loaded.
* If one or more lines in the file is not valid, nothing is loaded, and the old ACL rules defined in the server memory continue to be used.

Command : > ACL LOAD

***LPUSH*** - Insert all the specified values at the head of the list stored at key. If the key does not exist, it is created as an empty list before performing the push operations. When the key holds a value that is not a list, an error is returned.

It is possible to push multiple elements using a single command call just specifying multiple arguments at the end of the command. Elements are inserted one after the other to the head of the list, from the leftmost element to the rightmost element. So, for instance, the command LPUSH mylist a b c will result in a list containing c as the first element, b as the second element, and an as the third element.

Return Value - [Integer reply](https://redis.io/topics/protocol#integer-reply): the length of the list after the push operations.

Command - redis> **LPUSH mylist "world"**

(integer) 1

redis> **LPUSH mylist "hello"**

(integer) 2

***LRANGE*** - Returns the specified elements of the list stored at key. The offsets start and stop are zero-based indexes, with 0 being the first element of the list (the head of the list), 1 being the next element, and so on.

These offsets can also be negative numbers indicating offsets starting at the end of the list. For example, -1 is the last element of the list, -2 the penultimate, and so on.

Return Value - [Array reply](https://redis.io/topics/protocol#array-reply): list of elements in the specified range.

Command - redis> **RPUSH mylist "one"**

(integer) 1

redis> **RPUSH mylist "two"**

(integer) 2

redis> **RPUSH mylist "three"**

(integer) 3

redis> **LRANGE mylist 0 0**

1) "one"

redis> **LRANGE mylist -3 2**

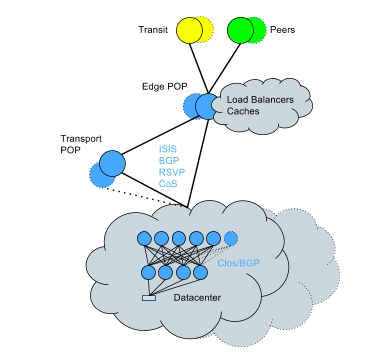
1) "one"

2) "two"

3) "three"

There are many more Redis Data structure commands, but the above-mentioned are the ones that are most frequently used.

**TRANSPARENCY**



**Fig 4 : Twitter Network flow**

By late 2010, They finalized their first network architecture which was designed to address the scale and service issues they encountered in the hosted colo. They had deep buffer ToRs to support bursty service traffic and carrier-grade core switches with no oversubscription at that layer. This supported the early version of Twitter through some notable engineering achievements like the TPS record we hit during Castle in the Sky and World Cup 2014.

Fast forward a few years and they were running a network with POPs on five continents and data centers with hundreds of thousands of servers. In early 2015 they started experiencing some growing pains due to changing service architecture and increased capacity needs, ultimately reaching physical scalability limits in the data center when a full mesh topology would not support additional hardware needed to add new racks. Additionally, existing data center IGP began to behave unexpectedly due to this increasing route scale and topology complexity.

To address this, they started to convert existing data centers to a [Clos](https://tools.ietf.org/html/draft-ietf-rtgwg-bgp-routing-large-dc-09) topology + [BGP](https://tools.ietf.org/html/rfc1105) – a [conversion](https://www.nanog.org/sites/default/files/20161016_Woodfield_Routing_Protocol_Migrations_v1.pdf) that had to be done on a live network, yet, despite the complexity, was completed with minimal impact to services in a relatively short time span. The network now looks like the figure above.

**SYNCHRONIZATION**

Manhattan is a general-purpose distributed key-value storage system that’s designed for small and medium-sized objects and fast response time. It’s one of the primary data stores on Twitter, serving Tweets, Direct Messages, and advertisements, among others. The primary goals behind building Manhattan were achieving predictability of runtime behavior, stability, and operational simplicity.

Every client request has to go through a Manhattan coordinator. A coordinator will group keys in each request by shard, and write the messages corresponding to per-shard operations into corresponding logs (e.g., “read key A” for shard 2, “check-and-set key B from 5 to 9” for shard 8). The coordinator also places its own callback address into the message to inform storage nodes about where to respond. The responses are necessary to provide results for operations like check-and-set, increment, and read. For strongly consistent writes, we only need to ensure that the operation is written to the log.

Next, storage nodes subscribing to particular logs will consume these messages, execute the operations one at a time, and respond back to the coordinator. When consuming logs, storage nodes always keep the position of the current operation on the log. They also have to atomically write that position to disk storage with the results of the corresponding operation. Otherwise, consistency could be violated during crashes, because some operations could be applied twice or not applied at all.

These per-log positions are also useful to check whether storage nodes that consume a given log are in sync with each other. When they respond to a coordinator, the coordinator can check whether the positions for a given operation match. If they don’t, this is an indication of data corruption having happened to this operation or its predecessor. Then we can find when the positions matched in the past and determine when corruption happened.

For performance reasons, the system has been decoupled. Twitter used to be fully synchronous. That stopped 2 years ago for performance reasons. Ingesting a tweet into the tweet API takes up to 145 msec and then all the clients are disconnected. This is for legacy reasons. The write path is powered by Ruby through the MRI, a single-threaded server, processing power is being eaten up every time a Unicorn worker is allocated. They want to be able to release a client connection as fast as they can. A tweet comes in. Ruby ingests it. Sticks it into a queue and disconnects. They only run about 45-48 processes per box so they can only ingest that many tweets simultaneously per box so they want to disconnect as fast as they can.

The tweets are handed off to the asynchronous pathway where all the stuff we’ve been talking about kicks in

**Consistency and Replication**

Hundreds of millions of Tweets are sent every day. They are processed, stored, cached, served, and analyzed. With such massive content, we need a consequent infrastructure. Storage and messaging represent 45% of Twitter’s infrastructure footprint.

Hadoop: We have multiple clusters storing over 500 PB divided into four groups (real-time, processing, data warehouse, and cold storage). Our biggest cluster is over 10k nodes. We run 150k applications and launch 130M containers per day.

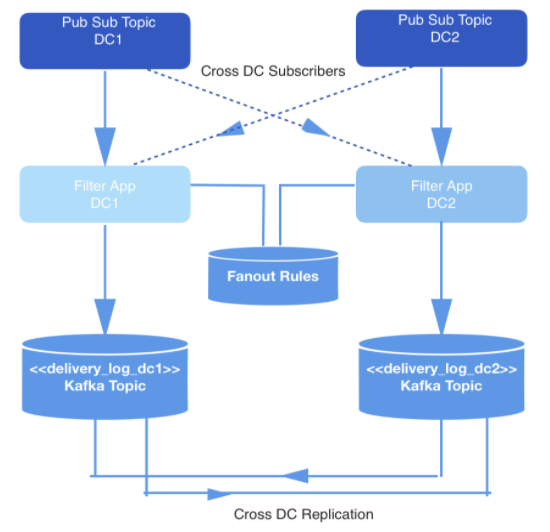
Hadoop/HDFS is also the backend to our Scribe-based log pipeline, but in the final testing phases of the transition to Apache Flume as a replacement in order to address limitations like a lack of rate-limiting/throttling of selective clients to aggregators, lack of delivery guarantee for categories, and to solve memory corruption issues. We handle over a trillion messages per day and all of these are processed into over 500 categories, consolidated, and then and selectively copied across all our clusters.

Ordinarily, when building a replay system that requires storage like this, one might use an architecture based on Hadoop and HDFS. Apache Kafka is used instead for two reasons:

* the real-time system was built on a similar pub-sub architecture
* the volume of events to be stored by the replay system isn't a petabyte-scale. We store no more than a few day’s worth of data. In addition, spinning Hadoop’s MapReduce jobs is more expensive and slower than consuming data on Kafka, which would not meet developer expectations.

To leverage the real-time pipeline and to build the replay pipeline by first ensuring that events that should have been delivered for each developer are stored on Kafka. We call the Kafka topic the delivery\_log; there is one in each DC. These topics are then cross-replicated to ensure redundancy which allows for serving a replay request out of one DC. These stored events are deduplicated before they are delivered.

On this Kafka topic, we created multiple partitions using the default semantic partitioning mechanism. Therefore, partitions correspond to the hash of a developer’s webhook, which is the key for each record. We chose a fixed number of partitions to spread out the data with the default partition strategy. With this, we mitigate the risk of unbalanced partitions and don’t need to read all the partitions on the Kafka topic. Rather, based on the webhookId for which a request comes in, a Replay Service (referenced below [Requests and Processing]) determines the specific partition to read from and spins up a new Kafka consumer for that partition. The number of partitions on the topic does not change because this affects the hashing of the keys and how events are distributed.



**Fig 5: Cross replication across Data Centers**

**Fault Tolerance**

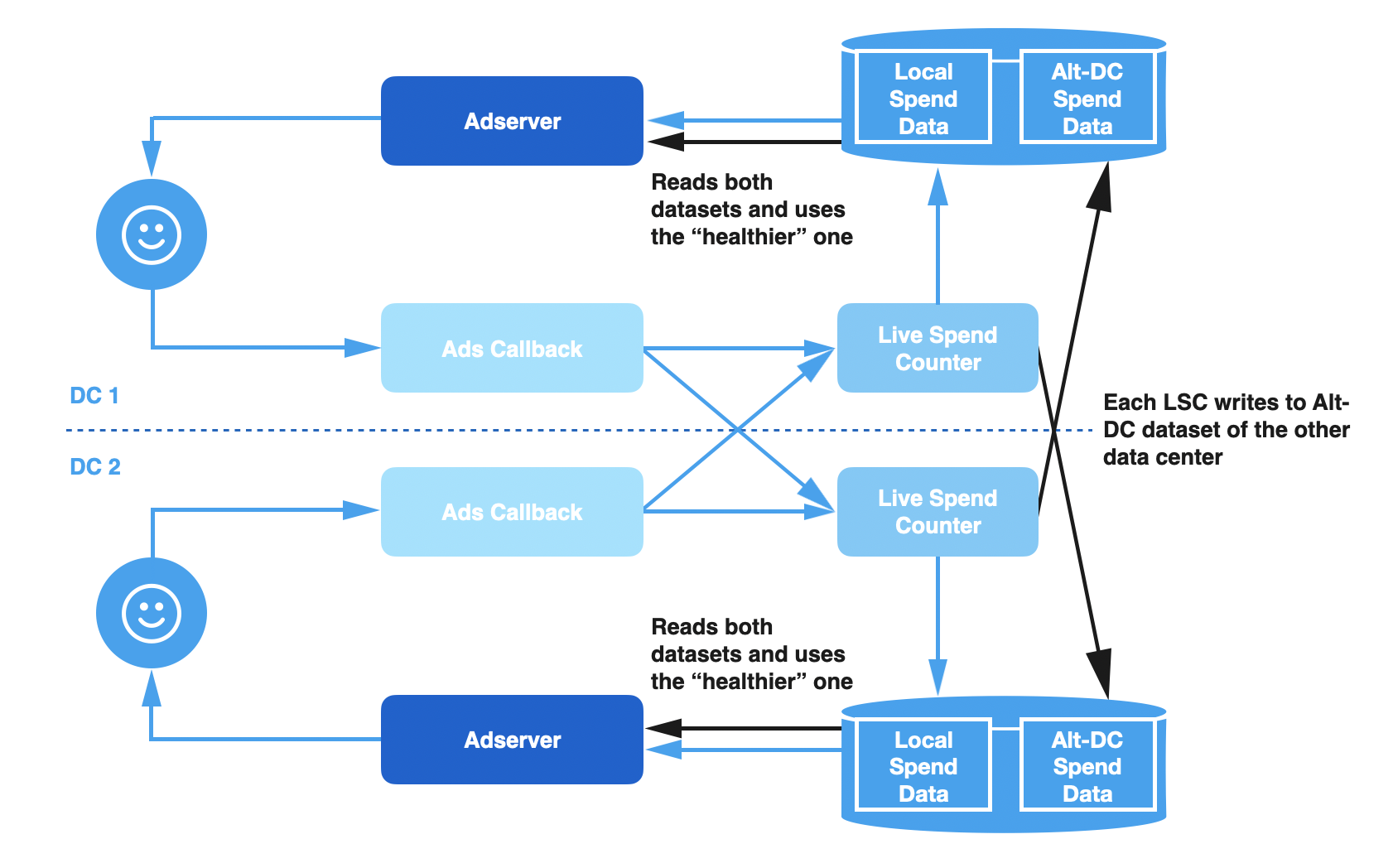
Although replicating engagement events gave us better consistency and accurate spend information, the system was not very fault-tolerant. For example, every few weeks, cross datacenter replication failures would cause the Spend Cache in the affected data center to go stale due to missing or lagging events. Typically, the Ads Callback pipeline would have system issues such as a garbage collection (GC) pause or unreliable network connection in one data center resulting in a lag of event processing. Since this is local to only a single data center, the Live Spend Counter in that data center would receive the events with a delay directly proportional to the lag, and as a result, the Spend Cache updates would also be delayed leading to overspending.

In the past, we mitigated these types of failures by disabling Live Spend Counter in the failing data center and configuring the healthy Live Spend Counter in the other data center to start writing to both its local Spend Cache and to the Spend Cache in the failing data center, until the lagging Ads Callback pipeline and Live Spend Counter were caught up.

Due to the many problems with this architecture, we re-designed our pipeline so that it could be more resilient to failure and require less operator intervention. The solution had two main components:

* **Cross-Data Center Writes**: the Live Spend Counter always writes to both the Spend Cache in the “alternate” data center as well as its own. It also writes some metadata about the health of that data. Each Live Spend Counter instance maintains two separate datasets, one computed with only local information, and one computed with the writes from the remote instance.
* **Dataset Health Check**: When serving an ad request, the Adserver pipeline reads both versions of the data, and automatically selects which one to use based on which dataset is healthier.

During normal operations, this system works identically to the previous design. However, if the local Spend Cache falls behind, the Ad Server is able to detect this and switch automatically to using the data set containing the writes from the remote data center. When the issue is resolved locally, the Ad Server will automatically switch back to using the local dataset since it is as healthy as the other one.



**Fig 6: Replication health check of two twitter data centers**

**SECURITY**

Twitter hires security professionals to do a threat model audit and/or penetration test. A good security firm will dig deep to uncover issues. Additionally, Twitter holds all partners accountable for the following:

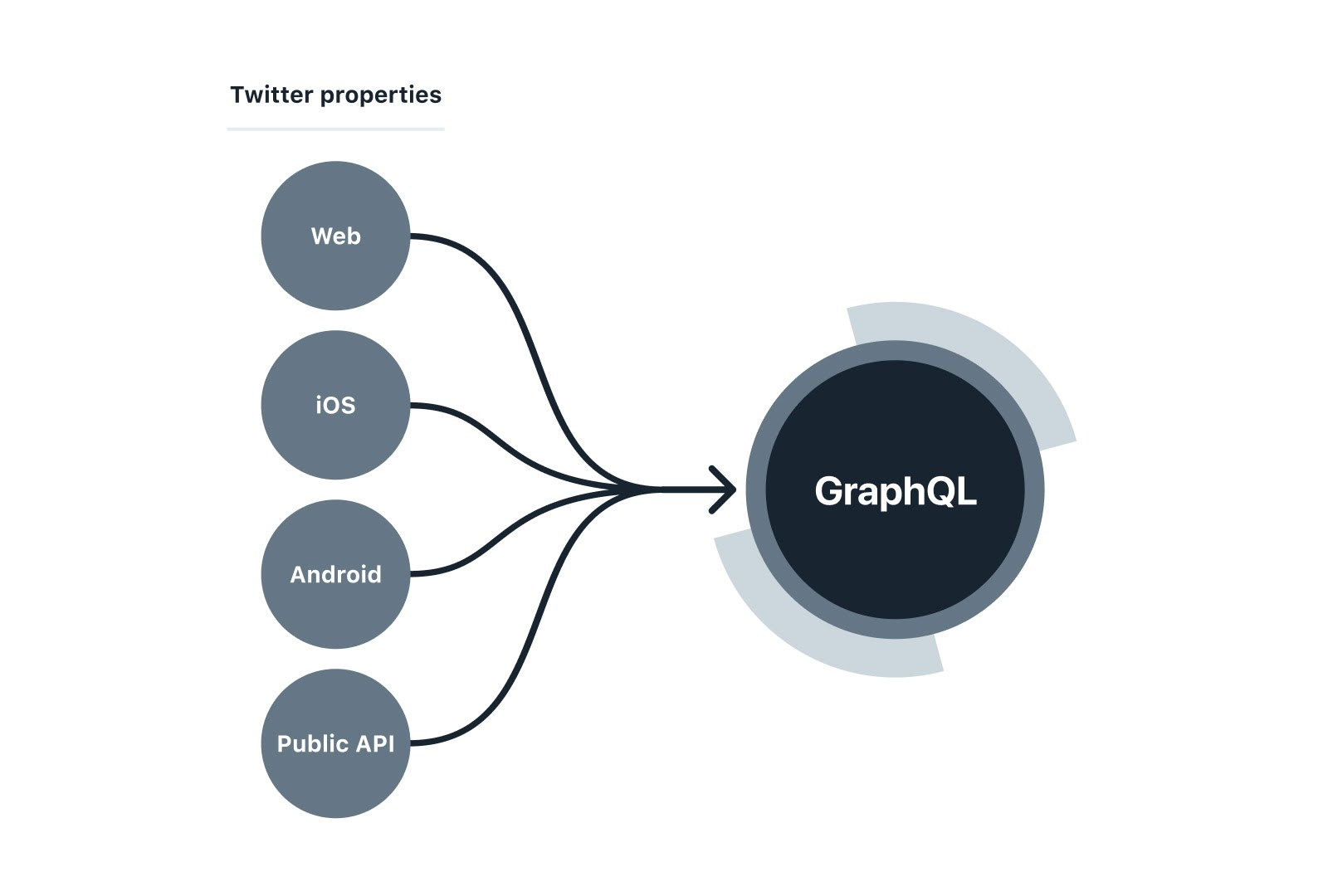
* Maintain code within a secure repository.
* Perform risk analysis throughout the Systems development life cycle (SDLC) process.
* Ensure security issues are identified and mitigated throughout SDLC.
* Ensure there exists segregation of environments throughout the SDLC process.
* Ensure all test defects are fixed, re-tested, and closed out.

On top of the usual confidentiality and integrity properties of HTTPS, forward secrecy adds a new property. If an adversary is currently recording all Twitter users’ encrypted traffic, and they later crack or steal Twitter’s private keys, they will not be able to use those keys to decrypt the recorded traffic.

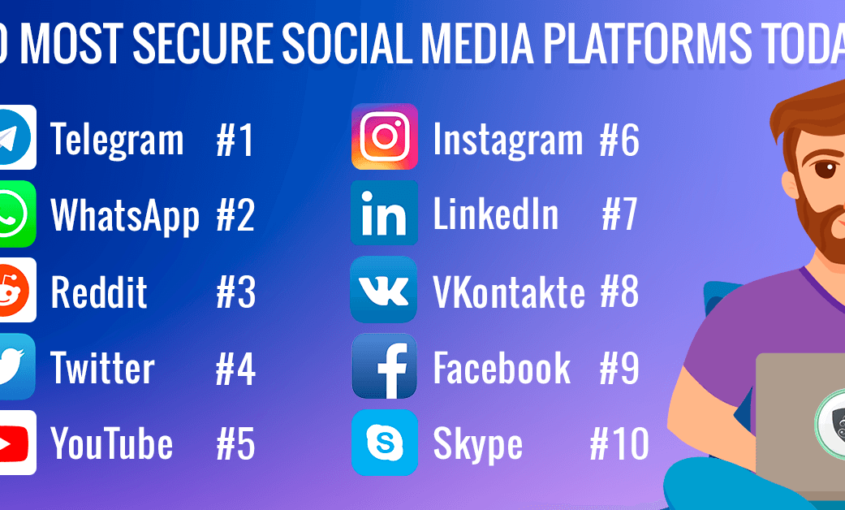
Under traditional HTTPS, the client chooses a random session key, encrypts it using the server’s public key, and sends it over the network. Someone in possession of the server’s private key and some recorded traffic can decrypt the session key and use that to decrypt the entire session. In order to support forward secrecy, we’ve enabled the EC Diffie-Hellman cipher suites. Under those cipher suites, the client and server management come up with a shared, random session key without ever sending the key across the network, even under encryption.

There are two main categories of Diffie-Hellman key exchange. Traditional Diffie-Hellman (DHE) depends on the hardness of the [Discrete Logarithm Problem](https://www.khanacademy.org/math/applied-math/cryptography/modern-crypt/v/discrete-logarithm-problem) and uses significantly more CPU than RSA, the most common key exchange used in SSL. Elliptic Curve Diffie-Hellman (ECDHE) is only a little more expensive than RSA for an equivalent security level.

* Representatives from teams building Twitter for web, iOS, and Android began migrating from individual internal REST endpoints to a [unified GraphQL service](https://about.sourcegraph.com/graphql/graphql-at-twitter)
* When an endpoint’s business logic can be represented in [StratoQL](https://drive.google.com/file/d/1aYupExDuAbUheDX4aycrxZDrc6stTcig/view) (the language used by Twitter’s data catalog system known as Strato which powers the GraphQL schema), then we only need to write a function in StratoQL without requiring a separate service.
* With the platform providing the common needs for all HTTP endpoints, new routes and resources can be released without spinning up any new HTTP services



**Fig 7: Platform compatibility with GraphQL**



**Fig 8: Twitter Statistics**

Due to the above-mentioned techniques, despite having a username password login for the portal, Twitter is considered to be one of the world's most secure social networking platforms.

**CONCLUSION**

Twitter is one of the readiest heavy systems with about 600,000 Tweets reads per second. To overcome this and serve its users it uses Redis which allows much faster access, but relying completely on Redis may cause problems hence copy of data is also stored in the database.

Twitter also provides Multi-platform support (i.e Twitter is available on android, IOS, Windows, etc) making it an all-inclusive platform

Load balancers are used to reduce the load on a particular node and twitter protects the data by using forward secrecy which is above the integrity properties of HTTPS.

Twitter is also politically persuasive, and hence most world leaders have an active Twitter account that they use to keep their audience engaged. Twitter has also grown its popularity due to various polls created by famous personalities that bring together a group of audiences that share a common interest.

Hence making Twitter a fast, safe, and secure distributed system.

**REFERENCES**

[1] <https://0x0fff.com/twitter-architecture-analysis-part-1/>

[2]<https://blog.twitter.com/engineering/en_us/topics/infrastructure/2017/the-infrastructure-behind-twitter-scale.html>

[3] <https://blog.twitter.com/engineering/en_us/a/2016/strong-consistency-in-manhattan.html>

[4] <https://blog.twitter.com/engineering/en_us/a/2013/forward-secrecy-at-twitter.html>

[5]<https://stackshare.io/twitter/twitter>

[6]<https://redis.io/commands>

[7]<https://www.youtube.com/watch?v=wYk0xPP_P_8>