Infrastructure

How we built Twitter's highly reliable ads pacing service

By
Eddie Xie
and
Yuanjun Yang
Thursday, 16 December 2021

Background

At Twitter and many other online advertising platforms, advertisers specify budgets for the campaigns they create. The advertising platform dynamically determines how the budget is spent throughout the day. Without careful consideration, a campaign can spend all its budget in seconds. This is not ideal, since it usually leads to suboptimal results and a narrow audience.

A pacing system distributes campaign budgets to maximize an ad's performance. In this blog, we describe how we separate Twitter's pacing system from the serving stack to an independent service. We offer an in-depth look at the technical challenges of building such a system and our solutions. Finally, we'll share the practical lessons we learned for designing a resilient service.

Why pacing service

Pacing is a critical component of any ad-serving pipeline. A rudimentary pacing algorithm was used at the beginning of Twitter's ads system. Over the years, the pacing algorithms were tuned and updated constantly.

Eventually, we encapsulated the pacing algorithms in a "Pacing Library" that runs inside the monolithic AdServer. The library computes pacing logic for each ad request that AdServer receives. We call this model request-based pacing. In the request-based pacing model, Pacing Library fetches the associated campaign's

budget and spend information and runs a series of algorithms to generate pacing parameters. The pacing parameters generated are then used by other modules in AdServer.

Two years ago, we started Project Tao to modularize each component in AdServer into separate services. Making pacing logic into a separate service brings a set of benefits.

First, we can easily run a time-based pacing model instead of a request-based pacing model. That means we can compute pacing parameters on the campaign level periodically, rather than for every ad request. It opens doors for state-of-the-art pacing improvement opportunities. For example, using a proportional-integral-derivative (PID) controller for pacing, reinforcement learning for pacing, and so on, requires such a time-based model.

Second, the new pacing service can accelerate feature development and algorithm improvements once it is decoupled from the centralized review process and deployment of the monolithic AdServer.

Lastly, it saves repetitive computation. Ad requests that originate from the same campaign have the same pacing parameters.

Because of the benefits mentioned above, we decided pacing is a great candidate for a standalone service.

However, being a standalone service comes with its challenges.

The most significant one is that the pacing service needs to be resilient (that is, “always-on”). If the service goes down, many other services might be impacted. This is because pacing parameters are constantly changing because they are computed based on a campaign’s spend and budget information.

Due to the dynamic nature of the pacing parameters, we cannot cache them or have a simple fallback strategy. For instance, we cannot return a set of predefined pacing parameters because they would not reflect the spend and budget status of a campaign. Using such pacing parameters interferes with the auction process, resulting in lost revenue for Twitter and suboptimal results for advertisers.

Thus, the pacing service needs to be a reliable service that can operate under many fault situations. To handle these situations, we bake in the resilience design from the beginning of the architecture design.

How we did it

To understand how we designed the pacing service architecture, it is helpful to first understand how the pacing logic fits into Twitter's ads systems.

How pacing works at Twitter ads

Pacing is a feedback loop system. At a high level, the diagram below shows how the pacing feedback loop fits the overall Twitter ads system.

The pacing service reads spend data from Live Spend Counter and budget data from Ads Database Service. The service does pacing computation every 10 seconds using the pacing library, which encapsulates all the pacing algorithms.

The computed pacing parameters then persist in storage for other services to use. If the ads from a campaign are served, the spend information is updated by Live Spend Counter, and pacing parameters for that campaign are updated.

Sharding design

The pacing feedback loop needs to be fast because campaigns can be served quickly, and their budgets can run out very quickly. When the campaign budget is spent too fast, the pacing service must be able to slow it down. Through experimentation, we found pacing performance to be optimal if we recompute pacing parameters for each campaign every 10 seconds.

That means we have only 10 seconds to finish computing pacing parameters for every campaign. However, at Twitter's scale, no one single instance can be fast enough for that requirement. Memory constraint for service instances also requires us to distribute the computation into multiple shards.

For sharding, we randomly bucket campaigns using 24-way hashing on account ID. Each shard contains three identical instances, shown below. Details of the sharding design are introduced in this related blog (https://blog.twitter.com/engineering/en_us/topics/infrastructure/2021/sharding-simplification-and-twitters-ads-serving-platform). We pick 24 shards to remain consistent with many other sharded Twitter ads services. However, there is no hard limit, and we can increase the number of shards if campaigns increase in the future.

Following the discussion above, one might wonder: if a campaign can only spend its budget by winning in an auction, doesn't its pacing status depend on other campaigns? If so, does the randomized sharding work without communication between the shards?

Indeed, the interactions between campaigns happen in the auction process outside of the pacing service. However, on the conceptual level, we can view the auction process as an implicit part of the pacing feedback loop. If a campaign wins in an auction and spends its budget, then in the next iteration (10 seconds later) of pacing computation, its pacing parameters are updated based on its updated spend data. Thus, the interactions between campaigns are propagated back to pacing service through updates of spend data.

This observation lets us greatly simplify the architecture design without introducing inter-shard communications.

Reliability design

As discussed earlier, one key requirement for pacing service is that it should have very high availability. At the same time, we need the pacing parameters to be published from a single instance to avoid having multiple controller updates. In the service design process, we studied success stories of other robust services and carefully customized them for the unique requirements of pacing service. We then collected a set of service failure scenarios and design modules that can mitigate those failures.

Instance failure and leader election

Instance failures are a common cause of service failures in a distributed system. As the number of instances increases, the likelihood of at least one instance failing increases.

To prepare for instance failures, for each shard we used two extra instances and formed a leader-election group. The leader election uses an in-house managed Zookeeper service.

For each shard, we specify a namespace in Zookeeper to keep track of which instance inside that shard is the leader.

```
/s/ads-pacing-service/shard-0
/s/ads-pacing-service/shard-1
/s/ads-pacing-service/shard-2
...
```

Within each shard, all three instances compute pacing, but only the leader can write its pacing parameters to the storage. This is desirable for the following reasons:

- Only the leader writes, so the cached data on the instance are consistent. Thus, the pacing parameters fluctuate less.
- The workers still read and compute throttle factors to keep the local cache warm and the internal state as close as possible to their leader. If a leader election happens, the pacing parameters fluctuate less.

The leader election ensures that as long as there is at least one functional instance, the shard is healthy. Twitter's instances are managed by an underlying resources management system. Dead instances are constantly swapped out and replaced. It is rare for three instances of the same shard to go down at the same time.

Zookeeper failure and manual override

At Twitter, Zookeeper is a managed service and has 99.99% availability. For most use cases, the SLA is good enough.

However, even with the high availability of the Zookeeper service, we still need to consider the following scenarios with decreasing likelihood of happening:

1. Zookeeper is available, but leader election takes too long.
2. Leader election results in none or multiple leaders.
3. Zookeeper is down.

In short, the pacing service needs to operate even when Zookeeper is not electing leaders correctly, or when it's not available. Given that this happens rarely, we used a simple design that serves as a safeguard, instead of a smart implementation that handles the scenario automatically.

To do that, we added a manual switch for each shard to override the leader elected by Zookeeper. We added monitoring to alert engineers when no leader or more than one leader is elected. When that happens, engineers are notified and

can manually intervene to make sure only one leader is running for each shard. Luckily, we have not had to intervene in two years of running the service.

DC failure and cross DC replicates

Twitter AdServer serves ad requests from multiple data centers (DC). In each DC, AdServer and other services read the pacing parameters from the local DC. Thus, we have an identical pacing service running for every DC.

The DCs are designed for resilience. We have dedicated teams for running and maintaining the DCs to make sure they are in a healthy state. However, some services that the pacing service depends on might have trouble in some DCs.

For instance, the Live Spend Counter service, which the pacing service draws on for spend information, might go down in DC1. Or the ads database service, which provides the budget information for pacing service, might be overloaded.

If the services that the pacing service depends on in a DC cannot recover within a reasonable period of time, the pacing service will have trouble computing pacing parameters in that DC. To prepare for the situation, we designed a cross-DC replication mechanism.
As shown above, each DC can write to another DC if we flip the switch. We intentionally design the switch to be a manual switch, because the failures happen so rarely that it is not economical to design a sophisticated automatic switching system. During the past two years of operating the pacing service, there was only one time that we needed to flip the switch to writing cross DC.

Time/Space trade-off and data prefetching

In addition to addressing reliability issues, we need to manage resources consumed by the pacing service.

Before pacing logic was running in AdServer, we ran pacing logic for each ad request. This introduces extra CPU time for each request and duplicates computation because many ad requests coming from a single campaign have identical pacing parameters. When we moved to the pacing service, we grouped all the pacing computations for each campaign together to avoid duplicate computations.

However, there were trade-offs when we moved from in-memory computation to remote fetching. The latter introduced latency and extra I/O usage.

The increase is multiplied when pacing is serving experiment traffic. To support pacing algorithm experimentation, each experiment needs its own budget and pacing state for the same campaign. And when other services are fetching for each experiment, the latency and I/O usage increase.

On the other hand, high-read traffic was a headache for pacing data storage. A huge capacity was needed to support the high-read queries per second (QPS) coming from thousands of instances of almost all ads services. This made pacing storage one of the biggest distributed storage clusters in Twitter. It was extremely expensive and hard to maintain.

Luckily, we were able to reduce more than 60% of the read traffic by optimizing the storage and local caching system design. Our storage is built on top of Redis, an in-memory data structure store. And by fitting the experimentation data into a B-tree and leveraging the Redis scan functionality, all pacing data for a single campaign can be packed into one remote response. Whenever the local cache on the client side receives a request, extra pacing data is also prefetched from remote and stored locally. Thus, data that are expected to be consumed in the near future are available locally even before requesting. Taking advantage of data locality helped us to reduce massive remote traffic and improve the stability of the local cache hit rate.

Lessons learned

Over the two years of designing, implementing, and operating the pacing service, we've learned a few things worth sharing.

First, **it is impossible to design a service for all faults and design for exhaustion.** Finding a balance between the complexities of fault tolerance and the likelihood of various faults is important.

For example, to reach the required SLA, we relied on data and back-of-envelope estimation for the likelihood of faults happening. Then we started tackling the most likely faulty events (instance failing) using proper design and toolings. For very rare events (happening less than once a year), we rely on simple designs and manual interventions. The simplicity of the system might help the overall health of the service in the long run.

Secondly, **building a reliable service pays off in the long run**. Over the past two years of operating the pacing service, we were able to maintain a healthy state of the service and deliver new features and improvements without constantly putting out fires.

Finally, the design phase matters. It might initially seem that time has been wasted on producing documentation instead of code. But the design feedback helps define the success of a project. In our case, our cross datacenter replication module was inspired by the success story of "How we fortified Twitter's real time ad spend architecture (https://blog.twitter.com/engineering/en_us/topics/infrastructure/2020/how_we_fortified_twitters_real_time_ad_spend_architecture)".

Impact and conclusion

After rolling out the pacing service, we noticed an immediate improvement in service maintainability and agility for development.

In the past two years, we were able to maintain zero incidents thanks to the reliability modules baked into the design and implementation. On-call issues happen rarely and are usually addressed within 30 minutes. As a result, engineers' efforts were freed from dealing with incidents and put to better use developing new features.

The simplified development and testing process also sped up building new features. Since shipping the pacing service, we are more efficiently experimenting for improvements.

This year, we start working on some big bets on the pacing service. These bets will provide advertisers more tooling to enhance their marketing strategy and improve their campaign spending efficiency.

We look forward to the exciting opportunities in front of us. If you are interested in solving these challenges, follow @TwitterCareers and come join the Ads ML Infra @Twitter (#JoinTheFlock).

Acknowledgments

This project took commitment, cross-functional alignment, and work from many teams. We would like to thank project leads who contributed to this blog: Eddie Xie, Yuanjun Yang, Bhavdeep Sethi, Lawrence Lam, and James Gao.

And others who worked on this project: Yong Wang, Smita Wadhwa, Yudian Zheng, Jiyuan Qian, Keji Ren, Kevin Xing, Hani Dawoud, Eitan Adler, Eric Chen, James Neufeld, Joe Xie, Juan Serrano, Paul Burstein, Su Chang, Srinivas Vadrevu, Udit Chitalia, Rachna Chettri, Yuemin Li, Radhika Kasthuri, Taj Darra, Hongjian Wang, and Neha Priyadarshini.

We would also like to thank the Revenue SRE team, Revenue QE team, Revenue Product and Engineering leadership team, and Revenue Strategy and Operations team for their constant support.

(https://www.twitter.com/eddiex)

Eddie Xie

(https://www.twitter.com/YuanjunYang)

Yuanjun Yang