Toward High Throughput Backend Provision for Mobile Apps with A Microservice Approach

Jen-Hao Kuo[1], He-Ming Ruan[1], Chun-Yi Chan[2], and Chin-Laung Lei[1]
[1]Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan
[2]Taipei Research Center, Shanghai Droi Technology, Taipei, Taiwan

For submission to The 18th Int'l Conf on Internet Computing and Internet of Things (ICOMP'17): LATE BREAKING PAPERS

Abstract—*Mobile Apps reveal a new era of IT industry by serving as the most important tools for modern people. To properly utilize the mobile Apps, App developers look for reliable backend support, and (mobile) backend as a service, which seems to be an elixir for this new epoch, emerges to fill up the vacancy. However, building such a system will be anything but a trivial task, since the behaviour of mobile Apps is no longer the same as traditional Web clients. In this paper, we propose a mobile backend framework by summarizing our experience of building a practical mobile backend service for mobile Apps, along with some experiments and field data from our testing site and production sites.*

keywords: mobile backend, microservice, high throughput backend service

1. Introduction

As mobile Apps becomes omnipresent in our daily life, the functionality an individual App can provide by its own can meet our needs no more. As a result, App developers seek out solutions for better resource provisions such as backend database or computation, however, maintaining a reliable backend system is a tough task for ordinary App developers. The eager for reliable mobile backend results in the arising of various backend services such as IBM DaaS, Google Firebase, and Amazon Mobile Hub. The essence of a mobile backend system is to take care of miscellaneous environment maintenance, including runtime environment, database, and all kinds of backend services, to provide a convenient and reliable resource delivery. Thus, App developers will be able to lay down the burden of backend maintenance and focus on their Apps.

Mobile Backend Service. A mobile backend service, which is sometimes so called mobile backend as a service (mBaaS), is a service model to provide all kinds of reliable backend features such as database access or push system to mobile Apps. Back to 2011, backend services, such as Parse, sprouted due to the explosive growth of smartphone users which encourages the society of mobile App developers and leads to the increasing need for mobile backend services [1], [2].

However, to the best of our knowledge, there is no academic research on mobile backend design and actual field report publicly available yet. Besides, there is still no dominating mobile backend service available on the market. Thus, the capability and feasibility of mobile backend service remains unknown.

1.1 Our Contributions

In this paper, we propose a mobile backend framework to provide reliable service with high throughput, implement the proposed backend system in microservice approach, and have some experiments on our proof of concept system and publicly available production system. Beside the framework, we also have some discussion about architectural design and issues encountered during actual implementations, in hope that our experience can benefit those who have plan to build their own mobile backend services or aim for doing similar researches.

1.2 Organization

This paper is organized as follows: related works can be found in Section 2; Section 3 illustrates the design of the proposed framework; experiment setting, test cases, and corresponding results are in Section 4; we have some discussion about mobile backend system in Section 5; and finally the conclusion of this paper is in Section 6.

2. Related Works

Mobile backend service is a relatively new domain and there is only few academic research available publicly, especially for framework design. However, there are still some relevant topics: microservice architecture, services front-end, service management, and backend service framework.

Microservice architecture. In traditional monolith architecture, a service is usually self-contained and consists of many components to fulfil its goal. Due to this natural of monolith architecture, any service upgrade/development will be painful since every change in any component will cause the recompile and redeploy the service. Besides, the compatibility issue will trap the developers in the already adopted technique simply because the better solution is not compatible with the old ones.

When it comes to micro service architecture, things are totally different. There is no all-in-one solution in microservice architecture, a service is broken down into several loosely coupled microservices tethered to each other by predefined API (usually RESTful API). As a result, with additional communication overhead between microservices, microservice architecture provide great flexibility for service development and deployment, improve stability by the synergy of loosely coupled and sometimes distributed microservice, and retain plasticity to adopt any new technologies.

To investigate microservice, articles by James Lewis and Martin Fowler [3] could be a good start. Sam Newman also gave a clear illustration about microservice in his book [4] in 2015.

Besides, there are also plenty works [5], [6], [7], [8], [9], [10] to investigate how to build service with microservice architecture with or without container.

Service front-end. To smooth the process of service acquiring, service front-end is a common solution. In 2005, Chen and Mohapatra [11] presented a framework to facilitate backend service via service brokers. [11] indicated that service brokers/front-ends can greatly benefit the management over services, including better QoS control, load control, cache control, and so on.

Service management. Service management including service deployment, monitoring, rescaling, failover, load balance, and other procedures required to operate a service.

We have plenty novel primitives for service management nowaday, for example, Prometheus and Zabbix provide customized metrics monitoring and alert system for service status monitoring; linkerd and namerd have accessible service discovery, load balancing, failure handling and other advanced feature by constructing a software defined network (SDN); DevOps tools such as Ansible and Jenkins simplify the service deployment procedure.

According to the environment a service inhabited, we can categorized the services into virtual machine/bare metal based and container based services. In general, container based service have several advantage over traditional virtual machine based service such as better performance [12], [13], [14], [15], agile deployment, and flexible resource control. However, container based solutions indeed have their drawbacks, for example, less isolation compared with virtual machines.

Thank to the concept of microservice and well developed primitives such as Docker and Kubernetes, container based service becomes more and more popular today.

Gropengießer and Sattler provide a framework [16] for virtual machine based database backend management. In [16], they have code generators to automatically generate service source code from human friendly schemas, deployment procedure to initialize the underlying virtual machine and the target service itself, and monitoring tool along with event-driven scaling tool.

Backend framework design. The design of the framework of a system almost dominates its service quality. Gessert et al. made their presentation about a framework of database backend in 2014 [17]. In their work, they regarded the unstable Internet as part of their backend system and aimed at improving the overall performance from the aspect of network instability instead of refine the database backend. As a result, they choose to reduce request latency of HTTP-based database query, and leveraged web caches to make their system more responsive. Although this approach shall greatly improve service quality, the confidentiality of request content remains as a limitation for mobile Apps, since the query and the response will not be public, especially for Apps with In-App purchase got involved.

3. Proposed Backend Framework

In this section, we will detail the proposed backend system, including the design of the proposed system, and how microservice adopted in our system.

3.1 The Proposed Framework

In this section, the we will detail the proposed framework for high throughput mobile backend system. In our system, we have three major components: 1.) service gateway, 2.) computing cluster, 3.) backend accessories:

- Service gateways: The service gateways check the validity of the connection layer content, extract the service layer information, and then forward the extracted information to corresponding computing cluster according to some predefined routing rules if the connection layer information is valid. After the request is properly handled, the result will be send back to the App via the service gateway.
- Computing clusters: The computing cluster is the essential component for request handling, all resource allocations should occur here. The computing cluster consist of 1.) **runtime engine** and 2.) **backend accessory front-ends**, where runtime engine handles various logic computations as needed and the backend accessory front-ends provides unified interface to corresponding backend accessories for the runtime engine. The runtime engine can further be divided into two categories: a.) **core functions** and b.) **Add-on functions**. Following a microservice approach, the term *function* implies that each function supports only an independent and basic functionality such as performing a database Read query. The difference between the core functions and Add-on functions is that the core functions provide the most fundamental functionalities such as CRUD database queries, permanent file storage, and push message; while the Add-on functions increase the flexibility of the mobile App developers by allowing them to design their own backend programs. However, the Add-on functions should subject to certain limitations for the sake of system security and performance. The benefits to provide such front-ends have been detailed in [11] and we will illustrate only those not covered in their work:
  - Issue addressing: If any unexpected issue occurs during backend accessory accessing in the system, the log from corresponding accessory front-ends will provide great help to locate the issue and track down the root cause, since the request for specific backend accessory will certainly pass through the corresponding front-end, and the front-end will be able to log the occurrence of issue by sending out access logs to the log cluster.
  - Agile request control: With such front-ends, we can easily cancel overdue request or mount rate limit on specific Apps. As a result, the backend accessories will be more reliable and the bussiness operating policy will also be more flexible.
  - Resource utilization: Using the accessory front-ends to force each access to the backend accessory simple and small, we can prevent that some resources are trapped into some bad operations, for instance, nested *$lookup* introduced in MongoDB 3.2 or native *forEach* operation on large collections can easily occupy the entire MongoDB for a certain period. Thus, using accessory front-end to enforce developers to access assorted resource in a predefined manner, which breaks a heavy task into several light tasks, can reduce this unwanted situation and greatly improve the resource utilization.
- Backend accessories: Backend accessories consist of various fundamental or value-add services, such as database, file storage, anti-virus scan, image processing, etc.

For a mobile App to access either the core functions or the add-on functions, a backend service could provide specific SDK with high level wrapper for core functions and low level function access interface for Add-on functions. Mobile App developers can then use the SDK to access the core functions if they need only the basic functionalities, or commit and use their own Add-on functions on the backend system if they want to make some non-trivial tasks on the server-side.

Figure 1 illustrates the lifecycle of a request for core functions in the proposed backend system.

3.2 Adopting Microservice Architecture

In our framework, we use accessory front-ends to define an unified and elegant service access interface for each backend accessory on the corresponding front-end. Besides, to adopt microservice architecture for a more robust system, we have four major direction of design: 1.) keep our API for backend accessories as simple as possible but with functional enough to support App developers for their customized logic, 2.) use runtime limitations such as execution time to enforce the developers to break down their tasks into lots of subtasks, 3.) teardown the primitive infrastructure into small services each with a single purpose, and 4.) under such condition, we can therefore have a great bunch of runtime replications as needed.

E.g., if an App developer want to crawl some open data and build his own database with certain expiration time on the backend, he/she will have to 1.) build a Add-on function as a spider to retrieve desirable information, and 2.) construct a routine job to do the housekeeping periodically. Thus, his/her App can access the up-to-date information summarized in his/her specific way via core functions or Add-on functions.

As a result, this design will increase the reliability of the system, since the tasks are divided and conquered separately, and the complexity of development and maintenance is reduced greatly.

4. Experiments

To verify the performance of the proposed system, we set up two environments, one testing site for stress test and one production site to serve App developers publicly.

4.1 Environments

Testing site. In our testing environment, we have six sets of computer nodes, detailed setting and spec of the nodes can be found in Table 1. And the topology of our environment can be found in Figure 2.

In our setting, we use Nginx 1.9.3 as our base to develop our service gateway, and use Apache JMeter to simulate our mobile clients. The reason we choose Nginx as our service gateway is that the non-blocking nature of Nginx can efficiently handle enormous incoming clients by reducing the impact caused by blocking condition, such as waiting for response from service provision servers or waiting for packet transmission. Thus, based on Nginx, we can forward vast amount of requests between our clients and service provision servers, and deliver a massive throughput.

For the service provision servers, we use CoreOS along with Kubernetes as our deployment environment, and customized OpenResty with LuaJIT as our runtime engine. Most of our backend accessory front-ends are written in GoLang, a novel programming language strike the balance between performance and accessibility.

Production site. In our production environment, we have seven servers with similar spec as our provision server in testing site, two of them serve as service gateways and the remaining five are used as provision server. Our production site located in Beijing, China, and users of our clients are almost comes from China (96.71%), and the remaining 3.29% are from all

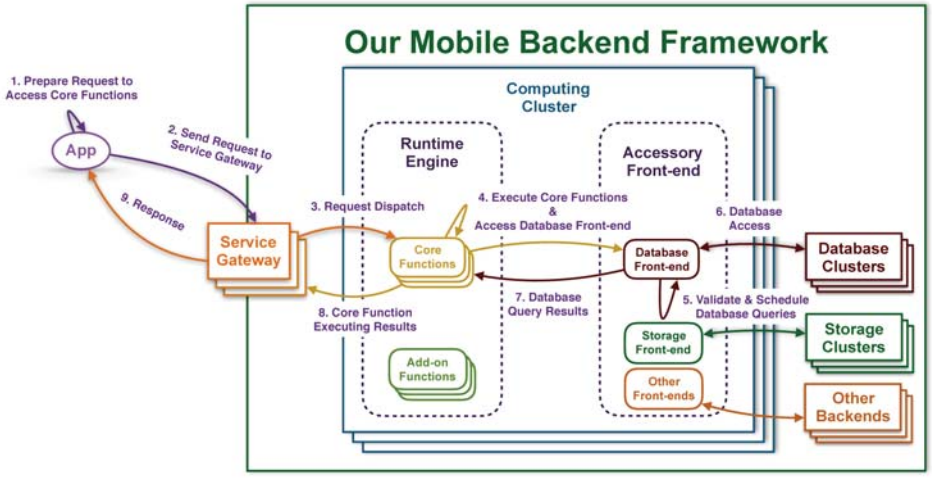


Fig. 1: Execution flow for core functions of the proposed mobile backend framework.

Table 1: Hardware/software environment of experiments.

Node	Hardware			Software	
	#Nodes	CPU	NIC	OS	Application
Service Gateway	1	i7-5960X	X540-AT2	Ubuntu 15.04	Customized NgX 1.9.3
Test Client	6	i7-4790	Gigabit Ethernet	Ubuntu 14.04	Jmeter 2.13
Provision Server	1	e5-2650v3 * 2	X540-AT2	CoreOS 1122	Computing Runtime
Database Cluster	3	e3-1270v5	X540-AT2	Ubuntu 14.04	MongoDB 3.2
Switch Type-A	1	N/A	10Gb Switch	N/A	N/A
Switch Type-B	1	N/A	Gb Switch	N/A	N/A

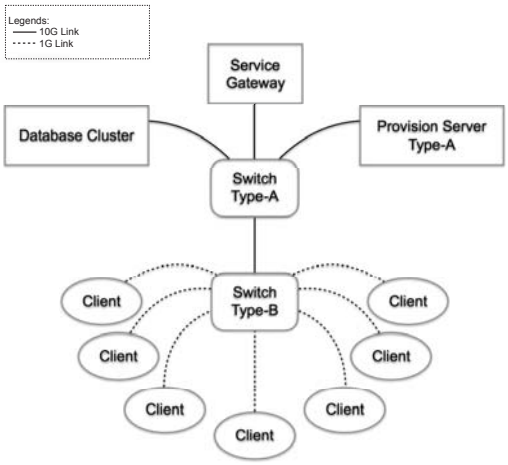


Fig. 2: Topology of our testing site.

over the world. With these hardware, we now support our customers with current request rate of 2000 request per second at rush hours without much effort (less than 5% CPU usage at service gateways and less than 20% CPU usage at provision servers). We now have 157 active Apps and 3 to 3.6 million daily active users which incurs 110 to 120 million valid request every day on our production site.

4.2 Experiments and Results

Testing site. To evaluate the performance of our framework, we have an experiment for end-to-end test of three cases: A.) Core function with database Read query, B.) Add-on function with database Read query, and C.) Add-on function performs simple arithmetic addition without database query.

To provide a baseline of our database capacity, if we query our MongoDB directly without additional access control, our MongoDB yields a request rate of about 60K request per second. Case A in this experiment follows the flow indicated in Figure 1; case B flows through Add-on functions instead of Core functions; case C makes a simple arithmetic operation at Add-on function and returns right after the execution. In case A and B, we make a query on an empty collection since we want to test the performance of our architecture instead of the performance of the database. Besides, due to the requirement of access and resource control over the database, requests in case A requires two to three additional query of ACL and authentication token to our MongoDB; while case B needs only one more query of ACL to the database.

We use JMeter with 1300 threads (250 * 6) to simulate the clients, however, we do not enable HTTP keep–alive, which means that each request will have to construct a new secure channel. As we can see in Table 2, with database query, the throughput will drop dramatically. The reason behind the phenomenon is that the built-in CRUD operations require three additional database queries for access control list for each database query compared to one additional ACL query of add-on functions.

Production site. In our production site, we collect field data from real world traffic of active mobile Apps containing 3,267,348,823 requests between 2017/03/01 to 2017/03/29.

App activity pattern. Figure 3 demonstrate a pattern of user activity, which is very consistent everyday. We have three peak time (7:00-8:00, 12:00-13:00, and 19:00-23:00) a day, and without much surprise, weekends have more request than week days.

Request elapse time. As we can see in Figure 4(a), the request execution

Table 2: End-to-end test of the proposed mobile backend framework on our testing site.

Case	Request per Second	Average Response Time (ms)
A	15.9K	91
B	23.9K	54
C	59.1K	23

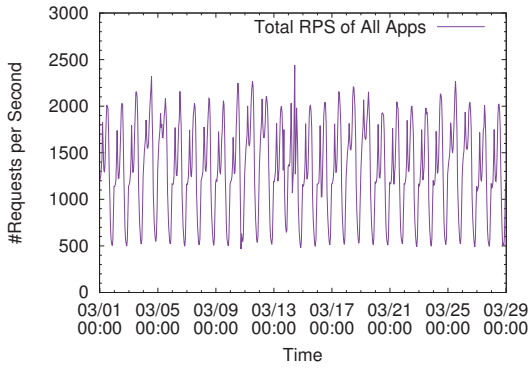


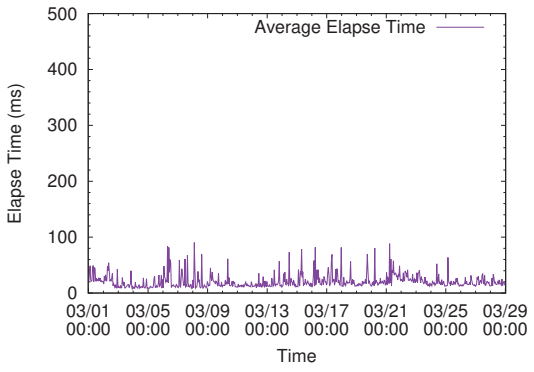
Fig. 3: Request rate of the proposed backend system on our production site.

time of our system remain stably under 100 ms, and less than 60 ms for most of the time. It is interesting that the timing of peaks of execution time seems to slightly match to the rush hours in the morning, however, the peak execution time almost occurs during weekdays. In the current stage, we consider that there coule be something to do with the user activity pattern.

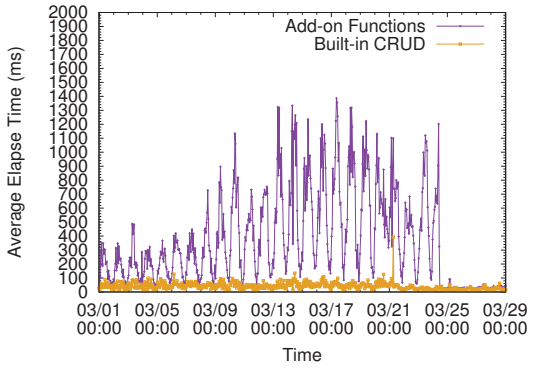
In Figure 4(b), we can find that the execution time of Add-on functions is pretty high, compared to overall average elapse time or average elapse time of built-in CRUD. What cause this phenomena is that we have only a few Apps use Add-on functions, and those Apps use their own Add-on functions to access their database and perform their own customized logics. However, after examine the database front-end, we find out that they query some specific fields frequently and these fields were not indexed until afternoon of March 24th. After applying proper indexes, the elapse time of Add-on functions reduce to similar level as built-in CRUD (less than 100 ms).

Request Composition. In our production site, we collect the access log of 3,267,348,823 requests from 2017/03/01 to 2017/03/29 to analyse the popularity of our services including but not limit to the following services: 1.) our proprietary protocol for secure channel validation, 2.) third party OAuth service to enable OAuth login for Apps using our SDK, 3.) App preference prefetch to fetch some App specific preference data, 4.) built-in CRUD for starndart NoSQL database access, 5.) analytic service to gather analytic data for App developers, 6.) over-the-air software update for partial or entire App, and 7.) Add-on functions for user defined backend logic. As we can see in Figure 5, among all the 3.2 billion requests, 28.47% of them are validation of our proprietary secure protocol. Except for our secure protocol and OAuth service, preference prefetch somehow become the most popular feature, which may related to the developing style of Chinese App developers, since we consider that it should only occurs every now and then.

To take a closer look, we use the built-in CRUD as the baseline in Figure 6, since we regard the database access as the most essential functionality of a mobile backend. Without much suprise, the population of Add-on functions is mere one tenth of the built-in CRUD request, since the developers are still not familiar to this new developing approach. However, we expect that in the near future, the ratio of Add-on functions will increase significantly, especially for those well-developed App companies, since the mobile devices are greatly limited by battery powers and use reliable backend service to take care of logic execution will benefit their reputation by providing a more reliable and battery friendly service.



(a) Overall elapse time.



(b) Add-on and core functions for CRUD

Fig. 4: Average elapse time of our production system.

The amount of analytic request is close to the built-in CRUD, which indicates that analytic data attractive to developers.

5. Discussion

Layered control. In our system, to block unwanted request as early as possible, we divide the request into two layers: **connection layer** and **service layer**.

- **Connection layer** follows certain proprietary or standard protocol to carry requests and responses to and fro between Apps and service gateways. The protocol the **connection layer** subjects to could require certain authentication information to authorize the App, for example, client-certificate for mutual authenticated TLS or request in JWT (JSON Web Tokens) form. As a result, information in **connection layer** can help the service gateways block disqualified requests and guarantees that requests reach backend service provision servers are issued by Apps using valid Software Development Kit (SDK). Besides authentication information, the **connection layer** also indicates where should this request be dispatched to.
- **Service layer** includes the actual requests and some authentication information from connection layer extracted by the service gateways. With the authentication information, the service provision servers can focus providing proper level of service/resource provision by merely checking the corresponding resource plan.

Take a HTTP-based request with TLS channel as an example, the range of connection layer is from the client-certificate to the encrypted TCP payload; while the service layer consists of the plaintext decrypted from the ciphertext and the identity of the App. Using this design, the service gateways can still have some degree of control over the incoming requests, since the service gateways can use the material for secure channel to validate if the request comes from a valid application. Thus, the service gateway can block plenty of unwanted requests without much effort and make the service

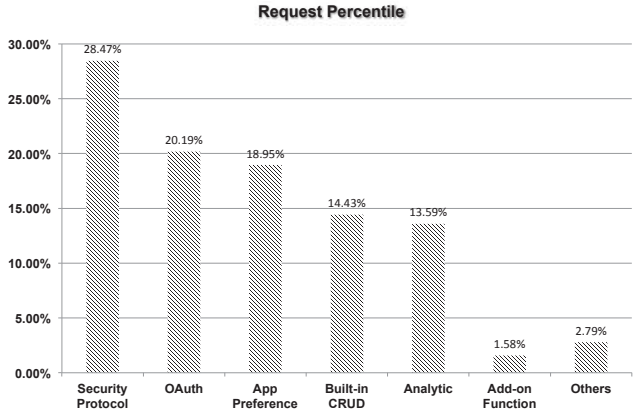


Fig. 5: Percentile of each request type on our production system.

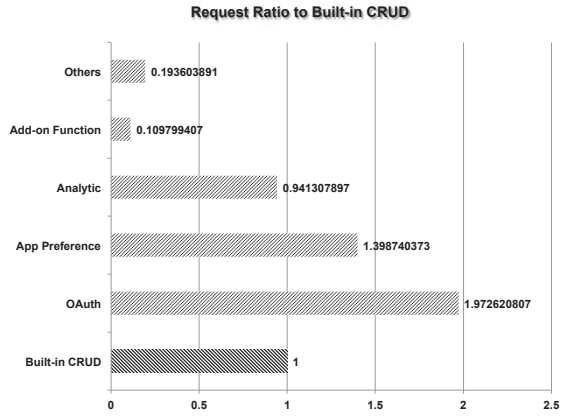


Fig. 6: Ratio of request amount to built-in CRUD request.

provision servers be able to focus on their business logic and resource control.

Thus, the service gateway can block plenty of unwanted requests without much effort and make the service provision servers be able to focus on their business logic and resource control.

Approaches for secure channel. Secure channel between service gateway and App is a must have primitive to secure the communication, and the most naıve approach could be adopting HTTPS or authenticated key agreement (AKA) [18], [19], [20], [21], [22], [23], which depend on Diffie–Hellman key exchange to negotiate temporary session keys and signature verification to prevent man–in–the–middle attacks. However, in our production environment, we find out that most connections between our system and our clients disconnect after delivering a single pair of request and response. Under such condition, secure channel using the traditional approaches above will become cumbersome for the service gateways. As a result, a more lightweight approach for secure channel construction becomes very crucial for a mobile backend system.

Containerized environments. To provide microservice, the combination of Kubernetes, CoreOS, and Docker, which is adopted by our system, is very popular now, however, the development of such system is very different from traditional virtual machine or bare metal environment. The most dramatic change from bare metal to Kubernetes could be the network environments. In Kubernetes, the default overlay network was once Flannel based on VXLAN due to the flexibility for deployment. However, this solution depends greatly on software (iptables) to handle packets and yields a very poor performance under large traffics (Case C of our experiment with Flannel gets only 20K request per second at most on the same hardware in our experiment environment, with very high CPU loading, especially software interrupt). As a result, we adopt Project Calico [24], a overlay network compatible to Container Network Interface (CNI), which is promoted by CoreOS to provide unified network plug-in for container environments, as our overlay network and get significant improvement of network performance (roughly 30K request per second on our initial attempts).

Database indexing. Take the case of Figure 4(b) in Section 4.2, we can realize the importance of database indexing without surprise. However, creating indexes will cost the database remarkable storage overhead, making indexes without limitation is never an option for service operators. Thus, except for provide developer-selected database indexing with proper limitations, auto-indexing is also a critical feature, since the developers might not understand the access behavior of their user while the backend system can easily obtain this information by analyzing the database access pattern.

Scalability. In the proposed framework, each component can be scaled out easily by adding new nodes.

- Service gateway: Since every service gateway is identical in our framework, we can simply add or remove service gateways by adding/removing physical service gateway and adjusting the load-balancer.
- Service provision servers: The service provision servers including computing clusters and backend accessory front-ends, which are deployed as pods of Kubernetes and can be scale out by simply operating the Kubernetes controller. Besides, if the workers are insufficient for the incoming requests, we can also add new hardware workers with the help of Kubernetes.
- Backend accessories: The scalability of the backend accessories depends on the solution adopted. To handle possible business growth, we recommend choosing solution with well scalabilities. Fortunately, most modern database clusters usually provide feasible scale-out solutions.

6. Conclusion

In the near future, we believe the mobile Apps will change our world permanently with the infinite possibility provided by the combination of mobile Apps and their backend services. However, to make it happen, there is still a long way to go, especially to make the backends accessible to App developers with high performance for better coupling between these two. In this paper, we proposed a framework of high throughput mobile backend system using microservice architecture, along with stress testing on our testing site and result from real world on our publicly accessible production site. Besides the experiments, we also have some discussion about mobile backend design and implementation according to our experience of building a public mobile backend system. With this paper, we wish to inspire more discussions to enlarge the society to draw the brilliant future of mobile Apps a little closer.

Acknowledgement

This study was supported by Taipei Research Center, Shanghai Droi Technology Co., Ltd. and by the Ministry of Science and Technology, R.O.C. under the Grants “MOST 104–2221–E–002–099–MY3”. We thank our colleagues from Taipei Research Center of Shanghai Droi Technology for their valuable insight.

We especially thank our colleage Jun-Yi Zheng and Chun-Han Tung who provided great help for our experiments.

References

[1] P. Sareen, “Cloud computing: Types, architecture, applications, concerns, virtualization and role of it governance in cloud,” *International Journal of Advanced Research in Computer Science and Software Engineering*, vol. 3, no. 3, 2013.

[2] I. C. S. Group. (2016) The 2016 state of dbaas report. [Online]. Available: http://www-01.ibm.com/common/ssi/cgi-bin/ssialias?htmlfid=CDM12345USEN

[3] J. Lewis and M. Fowler, “Microservices: a definition of this new architectural term,” *Mars*, 2014.

[4] S. Newman, *Building microservices*. " O’Reilly Media, Inc.", 2015.

[5] M. Abdelbaky, J. Diaz-Montes, M. Parashar, M. Unuvar, and M. Steinder, “Docker containers across multiple clouds and data centers,” in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, Dec 2015, pp. 368–371.

[6] N. Haydel, S. Gesing, I. Taylor, G. Madey, A. Dakkak, S. G. d. Gonzalo, and W. M. W. Hwu, "Enhancing the usability and utilization of accelerated architectures via docker," in *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, Dec 2015, pp. 361–367.
[7] D. Jaramillo, D. V. Nguyen, and R. Smart, "Leveraging microservices architecture by using docker technology," in *SoutheastCon 2016*, March 2016, pp. 1–5.
[8] H. Kang, M. Le, and S. Tao, "Container and microservice driven design for cloud infrastructure devops," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, April 2016, pp. 202–211.
[9] S. Patanjali, B. Truninger, P. Harsh, and T. M. Bohnert, "Cyclops: A micro service based approach for dynamic rating, charging & billing for cloud," in *Telecommunications (ConTEL), 2015 13th International Conference on*, July 2015, pp. 1–8.
[10] J. Stubbs, W. Moreira, and R. Dooley, "Distributed systems of microservices using docker and serfnode," in *Science Gateways (IWSG), 2015 7th International Workshop on*, June 2015, pp. 34–39.
[11] H. Chen and P. Mohapatra, "Using service brokers for accessing backend servers for web applications," *Journal of Network and Computer applications*, vol. 28, no. 1, pp. 57–74, 2005.
[12] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *2016 International Conference on Computing, Networking and Communications (ICNC)*, Feb 2016, pp. 1–7.
[13] A. Calinciuc, C. C. Spoiala, C. O. Turcu, and C. Filote, "Openstack and docker: Building a high-performance iaas platform for interactive social media applications," in *2016 International Conference on Development and Application Systems (DAS)*, May 2016, pp. 287–290.
[14] C. C. Spoiala, A. Calinciuc, C. O. Turcu, and C. Filote, "Performance comparison of a webrtc server on docker versus virtual machine," in *2016 International Conference on Development and Application Systems (DAS)*, May 2016, pp. 295–298.
[15] B. Varghese, L. T. Subba, L. Thai, and A. Barker, "Container-based cloud virtual machine benchmarking," in *2016 IEEE International Conference on Cloud Engineering (IC2E)*, April 2016, pp. 192–201.
[16] F. Gropengießer and K.-U. Sattler, "Database backend as a service: Automatic generation, deployment, and management of database backends for mobile applications," *Datenbank-Spektrum*, vol. 14, no. 2, pp. 85–95, 2014.
[17] F. Gessert, F. Bucklers, and N. Ritter, "Orestes: A scalable database-as-a-service architecture for low latency," in *Data Engineering Workshops (ICDEW), 2014 IEEE 30th International Conference on*. IEEE, 2014, pp. 215–222.
[18] H. Arshad and M. Nikooghadam, "An efficient and secure authentication and key agreement scheme for session initiation protocol using ecc," *Multimedia Tools and Applications*, vol. 75, no. 1, pp. 181–197, 2016.
[19] S. Blake-Wilson and A. Menezes, "Authenticated diffe-hellman key agreement protocols," in *International Workshop on Selected Areas in Cryptography*. Springer, 1998, pp. 339–361.
[20] L. Chen and C. Kudla, "Identity based authenticated key agreement protocols from pairings," in *Computer Security Foundations Workshop, 2003. Proceedings. 16th IEEE*, June 2003, pp. 219–233.
[21] H. Debiao, C. Jianhua, and H. Jin, "An id-based client authentication with key agreement protocol for mobile client–server environment on ecc with provable security," *Information Fusion*, vol. 13, no. 3, pp. 223–230, 2012.
[22] L. Law, A. Menezes, M. Qu, J. Solinas, and S. Vanstone, "An efficient protocol for authenticated key agreement," *Designs, Codes and Cryptography*, vol. 28, no. 2, pp. 119–134, 2003.
[23] Q. Xie, "A new authenticated key agreement for session initiation protocol," *International Journal of Communication Systems*, vol. 25, no. 1, pp. 47–54, 2012.
[24] C. Davenport, "Project calico," https://www.projectcalico.org, 2016, accessed November 2016. [Online]. Available: https://www.projectcalico.org