Lessons Learned in Applying Reactive System in Microservices

Alex Xandra Albert Sim[1], Okky Putra Barus[2], Frandy Jaya[1]
[1]PT Global Digital Niaga, Jakarta 10230, Indonesia
[2]Universitas Pelita Harapan Medan, Medan 20158, Indonesia

Because of the increasing growth of internet, internet services are moving towards microservice architecture for better performance and speed of development. Microservices do come with its own issue, mainly network I/O problem and resource utilization. One of the solution to that problem that also fits very well with microservices is reactive systems. This paper discusses about the experience in implementing reactive systems in microservice architecture, how it is done, and the result of the migration process.

Keywords: Reactive System, microservices, e-commerce, lesson-learned.

1. INTRODUCTION

Since the advent of ubiquitous internet, web developers have been challenged to handle big scale project, both in terms of traffic and project size. Developers must be able to build a web application that can handle big traffic since internet users keep growing year by year [1], while web projects keep getting bigger because of the ever-increasing features and requirements that needed to be deployed.

One of the answers to help developers in building a system that could handle a huge number of traffic without sacrificing maintainability and speed of development too much is microservice [2] [3]. As an architecture style derived from the Software Oriented Architecture (SOA), microservices brings maintainability, scalability, and ease of deployments to the table [4]. In recent years, enterprises and organizations that faced hyper-growth and big traffics have been increasingly adopting microservices to counter the issue [4].

Unfortunately, while microservices solved the problem it is designed for quite well, just like any other complex system it has its own problem that needed to be solved. For example, connecting (routing) the microservices is a challenge [5]. Another example is the network I/O and performance problems that naturally comes with network heavy systems like microservices [6].

Fortunately, solutions to the common problems in microservices have been tried and tested in both industries and academics. One of the proposed solution to the performance problem, for example, is reactive systems [7].

This paper is written to share the lessons learned and experiences both in building and adapting reactive systems to microservices architecture, specifically from the technical side. Experiences and lessons learned from the team-management and cultural side will not be discussed in this paper.

This paper is organized as follows. In section 2, the problems faced in production systems that triggers a discussion on improvement plans will be discussed. In section 3 will consist of a detailed thought process and explanation on why reactive system was brought in as a solution. The result of development and improvements that was seen after reactive systems will be shown in section 4. Finally, a conclusion and key lessons discovered will be shared in the final section.

*Email Addresses: alex.sim@gdn-commerce.com, okky.barus@uph.edu, frandy.jaya@gdn-commerce.com

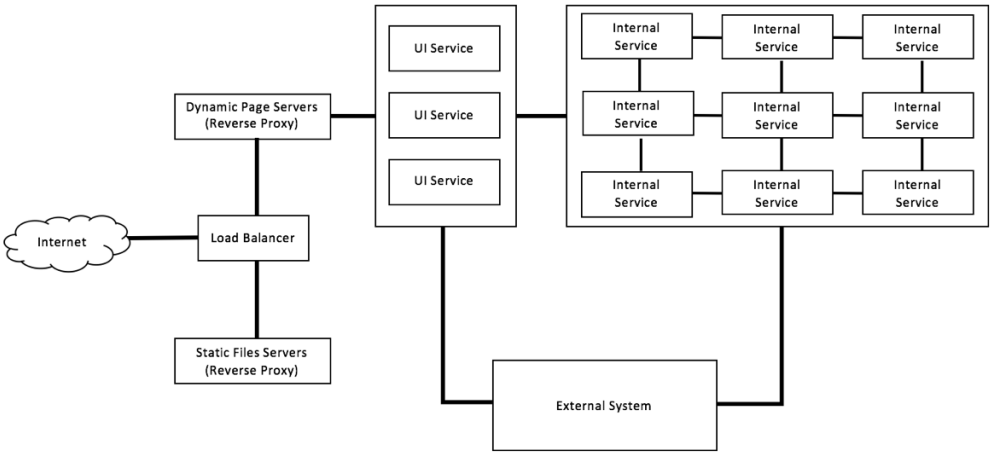

Figure 1. The Overall Microservice Architecture in Blibli.com

2. CURRENT IMPLEMENTATION OF MICROSERVICES IN GDN

The overall architecture of the primary service of PT Global Digital Niaga, blibli.com, can be seen in Figure 1. When a user comes into the site via HTTP or HTTPS, a load balancer would redirect the request to two servers. A request of static files would be routed through special servers that would only serve static files, and a request to dynamic content will be routed through a server that will connects the user to UI (User Interface) services. Both of these routers are reverse proxies that map a URI with server location.

A UI service is simply a microservice with a sole purpose of serving a UI to the user. Most part of the UI in blibli.com is served through a microservice, so that each part can be optimized based on requirements. For example, some part of the UI might need to be dynamically served with a SPA (Single Page Application), while the others might need a simple server-rendered page for SEO (Search Engine Optimization) purpose. A UI service might also access external services, i.e. the login page might have integration with external login service to logs user in with external credential.

All of the UI services will need to provide information that only the backend services have. These will need to call internal services. There are hundreds of internal services in blibli.com, each have different purpose and requirements. All communications from the UI services and among internal services are done via HTTP or HTTPS, with JSON as the data communication format. This means that the UI and internal services fundamentally are the same: they serve as a unit of computation that only does a handful of things related to their domain. The main difference between the UI and internal services are their output. The UI services almost always serves a text/html output, while the internal services serve application/json output. This is how blibli.com implements microservices: by separating concern of each requirements (e.g. search, viewing product information, executing purchase transaction) to a different internal and UI services.

While the UI services have a very simple architecture where it only serves static frontend files (usually compiled javascript), the internal service have a different architecture. Figure 2 shows the architecture of internal services.

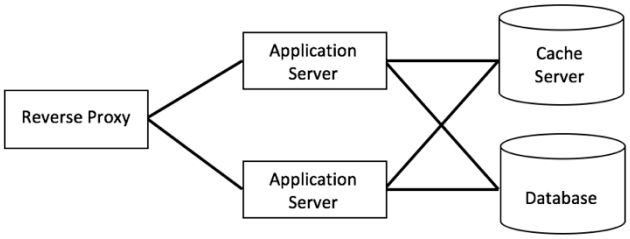

Figure 2 Internal Service Architecture

Each service has a reverse proxy that will be used as a point of access for the service. The reverse proxy then sends traffic received to application servers. One service has at least two application servers to achieve high availability, i.e. when one application server goes down or slow, the other will picks up the slack. The number of application servers itself depends on what the service does and how much it is accessed. The more calculation intensive it is, the more application servers it needs. There are critical services that have more than 10 application servers.

Internal service also usually has its own database and cache server. The kind of database and cache (i.e. relational, document-based, key-value) and how it handles scalability (i.e. clustering, load-balancing) will not be discussed in this paper as it is not relevant with the problem found in this architecture.

The microservice architecture implemented in Figure 1 has served blibli.com well, serving millions of pages per day but it is not without its problem. There are several problems that are found after implementing it production environment:

- Services that consumes external services face problems with reliability of the external services. If the external service is slow or is called via an unreliable network, the service will have performance problem.
- Some services need to consume multiple other services. When one or more of the services consumed have problem, it will cascade on to the caller.
- When not handled correctly, one service having performance problem could potentially cascades onto other services, and even bring the whole site down.

The three problems outlined could affect both the whole system's resilience and reliability, especially when there is a spike of traffic coming in, for example due to a promotion event.

When a service calls a problematic service and the callee took too long to give response, then the caller might exhaust its resource and halt, triggering a systemic failure. In the current architecture, all calls to services are done in a separate thread, and in effect, when the called service took too long, the number of threads in the application server would explode since new requests will keep coming and get served even when the service has not finished processing the current requests. It is also observed that this is the biggest problem coming out of the architecture. It boils down to a network I/O problem.

3. REACTIVE SYSTEMS

The problems outlined in Section 2 made it obvious that improvements are needed for the system. In particular, to fix all the problems, the improved systems have to:

- Handle failures and be available even during outages.
- Always be responsive even when other services it depends on are not.
- Be able to still communicate by sending or receiving messages in various network conditions.
- Works reliably even on varying or huge load.

Handling various load while not effecting other systems or performance is definitely a resource utilization problem. For example, to combat thread exhaustion, we could both use thread pool and limit the number of thread that's executed at any time. Then to make sure the system is not overloaded, we could also implement a timeout mechanism for all service calls. These techniques and tools are well-studied and well-known both in academics and industry [8] [9].

Being able to handle failures and respond even on outages and when facing an unstable network is what we define as being *responsive*. Again, this is an old problem that has been solved and even has various tools built around it. For example, Erlang is a programming language specifically designed around reliable distributed system that faces the same problem as outlined in Section 2 [10].

Seeing various tools and techniques available, there are several considerations in picking a solution to potentially solved the problem effectively:

- Since the current blibli.com is a big codebase, spanning millions line of code in hundreds of repositories, moving all services to a new system at once would not be possible. The solution needs to be able to mesh and implemented bit by bit on the existing codebase.
- Infrastructure change should be minimal and not disruptive to the current running application. Downtime is not tolerated as a business goal.
- There are hundreds of developers working in dozens of teams in GDN. The solution provided need to be familiar and easy to integrate to the current codebase.

Because of these constraints, the technology used will be the same as the current codebase: Java, both the language and the ecosystem. New features in the Java language and the Java Virtual Machine designed for reactive systems like streams, lambda, and asynchronous construct like Future make the choice to go with reactive systems easy. Anything supported by the language creator will have community and ecosystem leverage meaning there will be plenty documentations, libraries, and frameworks that will utilize the feature.

Integration is done first by building a new project that's not critical, in this case a system to track user's wish list and membership points. The project was built from scratch to be fully reactive, while still using the same framework and patterns from the legacy system. After successfully deploying those non-critical system, the development teams then are confident that they could utilize reactive systems to the whole codebase.

The first critical system to be implemented in reactive system is search and product details. The main reason implementing reactive system on those two systems is to improve performance. The current performance of search and product details page can be seen in Table 1.

Table 1 Search and Product Detail Performance Number

	Throughput (RPM)	95th percentile response time
Product Detail	2000+	10s+
Search	2500+	1s+

Note that these performance number are for normal days. When an event is held, the numbers could easily be an order of magnitude higher. And while the current system would be able to handle more traffic daily then peak traffic last year, analysis shows that the system will receive even more traffic this year for a promotional event. This prompts the need to move those two services to reactive system since a bad performance from those two could potentially take down the whole blibli.com.

4 RESULTS

There are three components of the output that were measured as the key to a successful system improvement for blibli.com. The first one is thread count. The system must be able to handle high loads and traffic without exhausting all the resources it has. As mentioned in Section 2, one of the biggest culprit in the legacy system is the thread number rising too much. Measuring and optimizing for thread count is then, important.

The second key performance indicator is throughput. As both internet users and GDN's customer base keep growing and growing, being able to serve more customer without adding more hardware will be very good for GDN as a business. This is why throughput is important.

The third important number for GDN is the 95th percentile response time. A faster site will fuel new customer growth and customer loyalty at the same time [11]. It is obvious that a faster site is an important business goal.

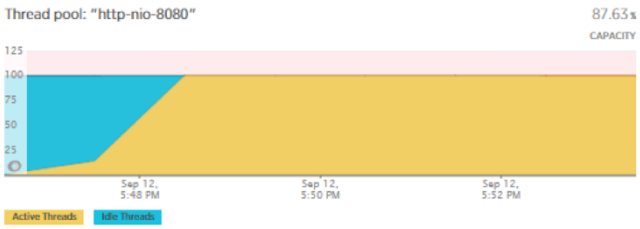

Figure 3 Thread Growth before Reactive Implementation

Figure 3 shows the system's thread pool count and growth over time on the search service in blibli.com before implementing a reactive system. As shown in the picture, without reactive systems the thread count went up to max thread (100) very quickly, exhausting all the resource available to the system and eventually brought the system down. The thread count after implementing reactive systems can be seen in Figure 4.

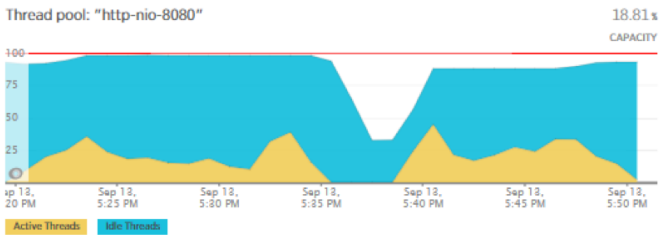

Figure 4 Thread Growth After Reactive Implementation

As shown in Figure 4, after implementing a reactive system, the resource utilization is now stable and goes up and down following the traffic. Whenever more traffic comes in, the thread count goes up, but still not exhausting everything. This is the effect of better scheduling provided by the reactive system's implementation.

For the next measurement, the throughput and response time comparison for last year's and this year's event can be seen in Table 2.

Table 2 Throughput and Response Time Comparison

	Throughput (2017 vs 2016)	Response Time (2017 vs 2016)
Product Detail	1.8x	3x
Search	5.7x	0.5x

There are obvious improvements in throughput and response time shown in Table 2. Note that in the case of search, there are decrement in response time, with dramatic improvement in throughput. This is due to throttling in both the search service and web service that's calling the search service resulting in additional latency. This is done deliberately, to make sure the search system doesn't have too much request since optimization work has not been done yet. The decrease in throughput is then, expected.

The response time increase in product details is the effect of calling all the services that it depends on asynchronously. Since reactive system enables developers to call services asynchronously with ease, the worst-case scenario in calling more than one service now become the slowest service instead of the sum of all services' response time.

5. CONCLUSIONS

One of the biggest problem found in microservices architecture is resource utilization, network I/O, and failure handling. Reactive systems help tremendously in all of those problems, while providing developers tools to work on optimizing their applications. While the improvements of performance in reactive system is not too dramatic, the improvements in resilience and stability due to better resource allocation should not be taken lightly. On an important production system, stability is often more important than performance, and reactive system does improve stability dramatically.

REFERENCES

[1] Cisco Systems, Inc., "The Zetabyte Era: Trends and Analysis," Cisco Systems, Inc., 2017.

[2] M. Villamizar, O. Gracés, H. Castro, M. Verano, L. Salamanca and R. Casallas, "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud," in *Computing Columbian Conference*, Bogota, 2015.

[3] A. Krylovskiy, M. Jahn and E. Patti, "Designing a Smart City Internet of Things Platform with Microservice Architecture," in *Future Internet of Things and Cloud International Conference*, Rome, 2015.

[4] N. Dragoni, S. Giallorenzo, A. Lluch-Lafuente, M. Mazzara, F. Montesi, R. Mustafin and L. Safina, "Microservices: Yesterday, Today, and Tomorrow," Open Science Framework, 2016.

[5] E. Casalicchio, "Autonomic Orchestration of

Containers: Problem Definition and Research Challenges," in *International Conference on Performance Evaluation Methodologies and Tools*, 2017.

[6] N. Kratzke, "About Microservices, Containers, and their Underestimated Impact on Network Performance," in *Conference: Cloud Computing*, Nice, 2015.

[7] K. Malawski, Why Reactive? Foundational Principles for Enterprise Adoption, Sebastopol: O'Reilly Media, Inc., 2017.

[8] V. Ghini, F. Panzieri and M. Roccetti, "Client-centered Load Distribution: A Mechanism for Constructing Responsive Web Services," in *System Sciences*, Maui, 2001.

[9] D. Xu and B. Bode, "Performance Study and Dynamic Optimization Design for Thread Pool Systems," Ames, 2004.

[10] J. Amstrong, "Making Reliable Distributed Systems in the Presence of Software Errors," Microelectronics and Information Technology, Stockholm, 2003.

[11] S. S. Srinivasan, R. Anderson and K. Ponnavolu, "Customer Loyalty in E-commerce: an Exploration of its Antecedents and Consequences," *Journal of Retailing,* 2002.

[12] J. Bonér and V. Klang, "Reactive programming vs. Reactive systems," Lightbend, Inc, 2016.

[13] D. Namiot and M. Sneps-Sneppe, "On Micro-services Architecture," *International Journal of Open Information Technologies,* vol. 2, no. 9, 2013.