Messaging Mesh as a Loosely Coupled Microservices Pattern

Ken Y. Chan
Innovation Lab, Timeleap Inc.
Toronto, Ontario, Canada
{ken, innovation-lab}@timeleap.com

Abstract: *This paper presents a novel microservices pattern called messaging or micro-messaging mesh (MiMH)[1] that extends the service mesh pattern using micro-messaging brokers. The micro-messaging brokers are specialized light-weight messaging brokers with small footprint providing synchronous, asynchronous, and/or semi-synchronous messaging capabilities; enabling loosely coupled communication among microservices via sidecar proxies. They orchestrate the message flows among microservices via sidecar proxies, and they can also coordinate the message flows among each other using REST-based messaging; thereby, reducing the control flow complexity embedded in the microservices and the sidecar proxies. Consequently, the messaging mesh pattern improves the reusability, extendability, resiliency, and scalability of microservices. That translates into a reduction in system complexity, leading to further cost saving and shorter time to market for an organization. Additionally, we have provided a reference implementation of MiMH as a framework[1]; it supports applications and processing of large data sets and volume across different business domains. To validate our MiMH framework, we have developed a proof-of-concept system[1] that evaluates and forecasts the exposure risk of COVID-19, as specified in the user queries, using publicly available COVID-19 data sets, social activity statistics, and user geolocations. This pandemic data analytic, and processing application and system are implemented using cloud native computing tools like kubernetes.*

Keywords: *Microservices, Service Mesh, Messaging Mesh, Micro-message, Micro-Messaging Broker, Sidecar Proxy, SOA, MOA, Enterprise Application Integration, Distributed Systems, REST API, JSON-REST, gRPC, AMQP, Docker, Kubernetes, Kafka, RabbitMQ, Java, Python, Golang, NoSQL, COVID-19, Pandemic, Big Data Analysis, Machine Learning, Geolocation*

OUTLINE

I. BACKGROUND ON SERVICE-ORIENTED ARCHITECTURE, MICROSERVICES, AND SERVICE MESH

Before we dive into microservices, service mesh, and our proposed micro-messaging architecture, we will first describe how the concept of service and its history came about. The concept of service was built upon the concept of reusable software components and interfaces in distributed computing. These concepts are not new. In the late 1980s, CORBA was the most popular distributed object and service architecture, particularly in Telecommunication

[1]This material can be used for non-commercial purposes under the CC BY-NC-SA 4.0 licence. Please contact the author concerning commercial usage.

and Financial Service sectors, but the popularity of CORBA did not last long. In the early 1990s Internet era, RPC frameworks such as Java RMI, XML-RPC, and SOAP became popular for building web services. Then, in the early 2000s, Service-Oriented Architecture (SOA) along with WSDL-SOAP and UDDI emerged as the most significant enterprise architecture, technology, and standards for web services, as in Figure 1. The proponents of SOA believed that it would save time, resource, and money by both reducing the complexity of enterprise systems and increasing the reusability and extendibility of existing assets. While many solution providers promoted it as the holy grail to almost every enterprise application integration problem known to IT, the prophecy has not been fulfilled.

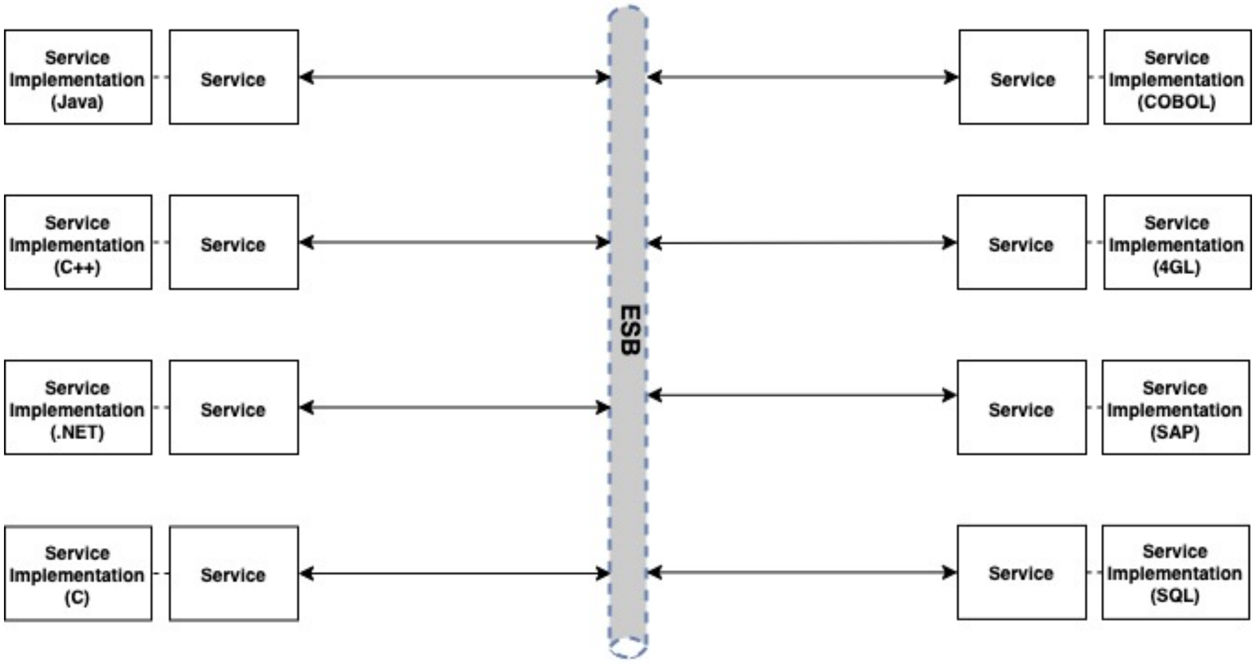

Service Implementation (Java)
Service
Service Implementation (C++)
Service
Service Implementation (.NET)
Service
Service Implementation (C)
Service
ESB
Service
Service Implementation (COBOL)
Service
Service Implementation (4GL)
Service
Service Implementation (SAP)
Service
Service Implementation (SQL)

Figure 1. Overview of SOA and ESB

There are three main problems with SOA in enterprises. First, many enterprises prioritize technology over problems. They tend to look for problems that a SOA solution could solve instead of the other way around. As a result, the IT team would end up forcing a complex SOA solution onto simple business problems. Secondly, most solution providers promote Enterprise Service Bus (ESB) as the silver bullet for SOA. Instead of having services to communicate directly with each other, all services communicate with each other through a bus [7]. A typical ESB implementation consists of many components such as: message brokers, message or service adapters, message transformers, service discovery, message queues, and persistent storage. Although ESB is conceptually simple, many ESB solutions turn out to be overly complex with many disjointed or overlapping components. It often takes a small army of specially skilled IT resources to build and maintain the stack. Unsurprisingly, many enterprises have limited their rollout of SOA implementations due to cost overrun. Thirdly, every enterprise stakeholder has a different interpretation of service, component, and application. Their interpretations often vary by the role, time, location, and mood of the individual. This leads to all sorts of planning and execution problems.

After the dot-com era, the money had dried up but the technology progression did not stop. Many technologists refocused their efforts on real user and technical challenges particularly in open source tools, communication protocols, and computing virtualization, instead of marketing hypes. In 2000, Dr. Roy Thomas Fielding from the University of California introduced the web community to the concept of Representational State Transfer (REST) in his PhD dissertation [1]. He proposed a set of architectural styles and network-based software architectures that would improve the modularity, reusability, extendibility, scalability and development speed of distributed systems. By the mid 2000s, the software development community had built upon his work and started to embrace the concept of REST-service and microservice architecture.

In the past few years, Open API and REST microservices became the new hot topics in the IT industry. Many influencers believed that they are the new holy grail of scalable and reusable software in distributed computing. As a matter of fact, microservice architecture is another variant of the service-oriented architecture (SOA) structural style that arranges an application as a collection of loosely coupled microservices. Although the concepts of REST microservices, and API are often mentioned as an inseparable bundle, they are actually mutually independent. Microservices are organized around business capabilities and a microservice should do one thing and do it well. Microservices are small in size, messaging-enabled, bounded by contexts, and autonomously developed. They are also independently built, deployed, decentralized, and released with automated processes [2, 6]. They should communicate using technology agnostic and lightweight protocols in a low-latency network. Last but not least, microservice-based architectures facilitate continuous delivery and deployment.

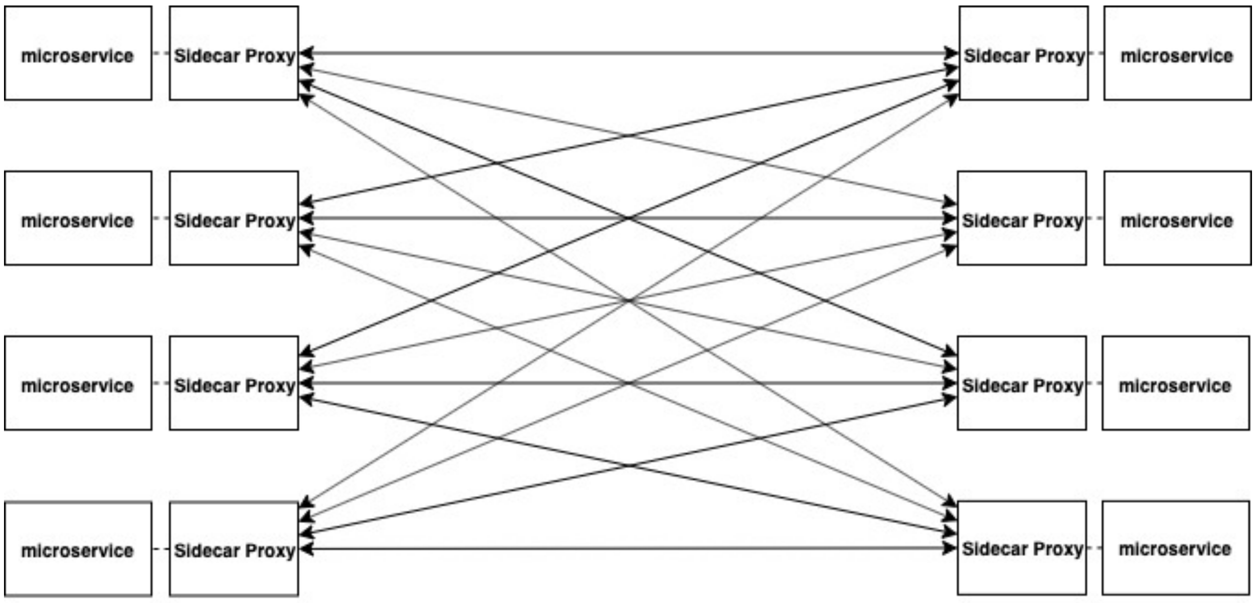


Figure 2. Overview of Service Mesh

Most recently, the concept of service mesh, as in Figure 2, has emerged as a popular variant of the microservice architecture. It is becoming a critical component of the cloud native stack [3, 6]. A service mesh is a dedicated low-latency infrastructure layer designed to handle a high volume of network−based interprocess communication among microservices using lightweight application programming interfaces (APIs). High-traffic companies like PayPal, eBay, Walmart, and Google have all implemented their own versions of service mesh in their production applications.

In practice, a service mesh is typically implemented as an array of lightweight network sidecar proxies that are deployed alongside application codes, without the application needing to be aware [3, 6]. A sidecar proxy is different from an API gateway (e.g. MuleSoft, Apigee). An API gateway acts like a centralized hub that exposes external API to the API consumers outside the local network and translates the inbound API requests into the internal API requests, and vice versa. An API gateway tends to emphasize API translation, security, and management for the service endpoints. On the other hand, a sidecar is a decentralized proxy that uses lightweight APIs to facilitate communications between microservices. It takes care of the non-functional aspects such as discovery, routing, and telemetry for the associated microservice and enables microservices to be more reusable. Since microservices can communicate directly with each other via their sidecars, they could also become quite chatty in a mesh configuration. The volume of network traffic in a service mesh will likely be much higher than a classic SOA. Therefore, service coupling, responsiveness, and throughput are big considerations in the design and implementation of a high-performance service mesh.

II. What is a Service, Software Coupling, Reusability, and Extendibility?

A distributed system comprises a number of services and components across trusted or untrusted environments. A *distributed application* provides end-user capabilities as a logical collection of services and components across distributed systems. So what is a service? A service is a software that offers its clients (consumers) with access to a set of software capabilities with a specific purpose through a prescribed interface. The software capabilities are typically associated with a set of operations and resources, particularly with specific business activities. Services are provided to other components by application components through a standardized set of communication protocols over a network. The standardized set of protocols is typically implemented as a well-defined interface for a specific purpose that facilities the reuse of a service. Additionally, a service has the following properties: (i) it should present a business activity; (ii) it has a well-defined interface such as API; and (iii) it should be a self-contained black box for its consumers which does not need to be aware of its inner workings.

As enterprise customers and users continue to demand higher quantity and quality of digital services, enterprises struggle to build applications and services efficiently enough with their existing digital platforms. Enterprises need more reusable and extendible software assets and better enterprise application integration platforms to accelerate their system development, while they improve the system quality and keep the costs down. Unfortunately, many implementations deviate from these principles. Many software assets often have overlapping or conflicting purposes, and a lack of regular maintenance. When these software assets are integrated together, they may provide unintended behaviours (outputs) that are beyond their original scope or specifications. Over time, the whole platform grows in every direction and becomes costly to maintain.

So how difficult it is to reuse or extend an existing piece of software? There is no exact definition for software reusability and extendibility. A software asset is built upon a set of requirements and specifications. A new application may reuse this software asset provided that its specifications satisfy the requirements of the new application. If not, the existing software asset may be modified or extended to meet the new requirements. The change must be carefully analyzed such that there is no impact to the existing system as a whole. Having said that, every software can technically be reused or extended for another purpose. The key deciding factors are the amount of effort and the time it takes to reuse or extend an existing software, versus building a new piece of software for a new application. Many research projects have shown that the amount of time and effort to reuse or extend a software for a new application is related to the complexity of inter-dependencies or couplings between software components. In fact, the amount of time and effort grows non-linearly or even exponentially as the software complexity increases. *Software coupling* has been proven as a key contributor to software complexity [4, 5]; hence, it is also a main roadblock to software reusability and extendibility.

So what is software coupling? Generally speaking, when *two pieces of software have either a dependency on one or the other, or a mutual dependency*, they become coupled [4]. A software coupling occurs when a piece of software has read or write access to the behaviour or the state of another piece of software in its source or at run-time; any change to the depended software may impact the behaviour and the state of the dependant software. There are many types of software couplings; however, the most concerning couplings are *control coupling* and *data coupling*. Control coupling is when one software module controls the flow of another module by passing it information on what to do. Data coupling occurs when modules share data through procedural parameters or global data space. Moreover, all couplings can happen either in the source (statically) or at run-time (dynamically) [4].

A software coupling or dependency is not necessarily a bad thing and does not necessarily have negative impacts on the reusability of the software. A software using or depending on another piece of software is often a sign of software modularization and reuse. This happens in all software development projects. On the other hand, when two pieces of software assets have an unintended coupling that is beyond their original scope or specifications and exhibit *side-effects* [11] that reduce their reusability and extendibility, then this is a *negative software coupling*. If a change were made to a negative software coupling, it would also lead to a substantial change to the other pieces of dependent software. In fact, this is a common scenario especially with a poorly designed and implemented software, which is often a software that does too many things and does not do them well. When multiple pieces of these software assets are integrated together, they often exhibit *unintended behaviours*.

Over time, the existing platform would become exponentially more difficult to extend and manage. The challenge is to reduce the amount of effort a software developer would need to reuse existing assets, thereby allowing the software developer to deliver the system with better productivity and a higher quality than existing approaches. These negative couplings can happen to a monolithic application, a microservice, or any software. Unfortunately, there is no magic tool that can eliminate all negative couplings in an existing system. The problem of software coupling is technology agnostic. This should be mitigated by a combination of well-defined scope and system specifications, architecture, design patterns, best practices, and appropriate technology selections.

III. Problem and Context Definition

Software coupling is one of the many contributing factors affecting the reusability, extendibility, responsiveness, resiliency, elasticity, and scalability of a distributed system [5]. Therefore, it is important to understand the impact of service coupling on microservices. One problem is that there are no definitions on the relationship between service coupling, reusability, and extendibility. There are also no specific guidelines on how to define and build loosely coupled microservices. Another problem is that a developer tends to take coding shortcuts; he/she may unintentionally or intentionally couple the behaviours; or worse, the internal state of the microservice via API and sidecar proxy; thereby creating a high degree of software couplings. Another problem is that most organizations tend to adopt SOA frameworks in conjunction with Message-Oriented Architecture (MOA) consisting of web services that mediate the legacy systems by using a mix of synchronous, semi-synchronous, and asynchronous APIs and messaging infrastructure. However, these SOA frameworks tend to be complex, require high overhead, and fail on reusability, and extendibility of the services. On the other hand, microservices need to communicate frequently and responsively. They are often deployed in small footprint virtualized containers that are distributed across environments with variable degrees of connectivity and trust. Therefore, existing SOA frameworks are not suitable for microservices.

Another problem is that the microservices pattern in general has not lived up to its early promise of high performance, efficiency, resiliency, scalability, reusability, and extendibility [3, 6]. Most recently, service mesh is one instance of the microservices pattern that is intended to fulfill these promises with its integrated data and control planes [3, 12-14]. However, the mesh communication model is not necessarily efficient for many interaction patterns. Furthermore, a service mesh places too much emphasis and complexity on the sidecar proxies; the scope of sidecar proxy is not well-defined. Hence, service mesh has yet to fulfill its promises as well. In addition, the existing service meshes, such as GCP Anthos Service Mesh with Istio [12], AWS Service Mesh [13], and Azure Service Fabric Mesh [14], have an overly complex control plane [3, 6].

Another problem is that there is a need for a flexible and loosely-coupled microservices architecture that supports popular technology platforms and stacks. This not only minimizes integration effort but also avoids reinventing the wheel. Finally, there are many topics related to software coupling such as cohesion, dynamic coupling of object, class, and relation [4]. Our work focuses primarily on control and data couplings of microservices either in the source or at run-time. Additionally, other related computing topics such as requirement engineering [4], feature interactions [8-10], and side-effects [11], also address these concerns. However, these topics and others are outside the scope of this paper.

IV. Our Contributions

This paper presents our three contributions that simplify enterprise application integration and the development of distributed applications and loosely coupled microservices. First, we provide a pragmatic definition of software and service coupling so we can assess the effectiveness of our techniques.

Secondly, we present our novel loosely coupled microservices pattern called *messaging mesh or micro-messaging mesh (MiMH)*[1] that extends the service mesh pattern using micro-messaging brokers. The micro-messaging brokers are specialized light-weight messaging brokers with small footprint. They provide synchronous, asynchronous, and semi-synchronous messaging capabilities, and also orchestrate messages among themselves and microservices using our REST-based messaging called micro-messages. Therefore, they enable loosely coupled communication among microservices. Our messaging mesh pattern enables high performance distributed systems and applications to be assembled with loosely coupled microservices using micro-messaging brokers; and hence, it can support applications and processing of large data sets and volume across different business domains. It is also platform, technology, and protocol agnostic. Therefore, our messaging mesh pattern *(MiMH)*[1] improves the reusability, extendability, and scalability of the microservices.

[1]This material can be used for non-commercial purposes under the CC BY-NC-SA 4.0 licence. Please contact the author concerning commercial usage.

Thirdly, we have developed a proof-of-concept (PoC)[1] system that allows a user to query for the exposure risks of COVID-19 using data sets from publicly available data sources based on geolocation criteria. The PoC is a data processing and analytic system that is built upon our MiMH framework. It can forecast the risk of exposure by analyzing publicly available COVID-19 data sets, social activity statistics, and user geolocations. Our MiMH framework supports protocols like HTTP, JSON-REST, gRPC, and AMQP, plus popular messaging solutions like Kafka and RabbitMQ. Most importantly, this PoC demonstrates that our MiMH framework can improve the reusability, extendibility, performance, scalability, and resiliency of the microservices.

V. Overview of Service Coupling

First, we define the concept of service coupling and its relationship with reusability and extendibility. Service coupling is a form of software coupling. By applying the definition of service coupling to our micro-messaging mesh pattern (MiMH), we enable MiMH to achieve loose couplings among microservices. So, what is service coupling? It is a form of service dependency that happens when a service consumer depends on the interface (behaviour) or the state of a service through a service provider [4, 5]. There are three dimensions to service dependency, as in Figures 3-6 and 7-10: *category*, *type*, and *degree*. With the category dimension of service dependency, we define two categories of service dependency: structural dependency and execution dependency. In the case of a structural service dependency, a service consumer needs to reference the interface or the state of a service at the structural level, such as the source code, configuration, or property file. In the case of an execution service dependency, a service consumer needs read or write access to the interface or the state of a service through a service provider at execution-time. Our work examines service couplings from both control and data aspects and in structural and execution dependency perspectives [4].

Regarding the *type* dimension of service dependency, there are also four types of service dependency: *state, function, endpoint,* and *resource* dependency. In the case of state service coupling, as illustrated in Figures 3-4, a service exposes its internal state typically as message parameters or content to the other services: it is a form of control and/or data coupling. In the case of function service coupling, as illustrated in Figures 5-6, a service needs to explicitly reference the function of the target service in the message: it is a form of control coupling. In the case of endpoint service coupling as illustrated in Figures 7-8, a service needs to explicitly reference the endpoint of the target service (e.g. URL) in the message: it is a form of control coupling. In the case of resource service coupling, as illustrated in Figures 9-10, a service needs to explicitly reference the target resource (e.g. URL) in the message: it is a form of data coupling.

From another aspect, the *direction* and *distance* of the service dependency may or may not affect the *degree* of the service coupling. A service is said to have a direct dependency or coupling on another service when it has an explicit reference or access to the interface or internal state of another service, as illustrated in Figures 3-8. On the other hand, a service is said to have an indirect dependency on another service, as illustrated in Figures 9-10, when it has a direct dependency on an intermediary which in turn has a direct dependency on the another service. The number of hops determines the distance of the service coupling. In both cases, any change to the service provider may impact the behaviour and state of the service consumer. Any impact that increases the amount of effort on making changes for the purpose of reusing the underlying software would reduce the overall reusability and extendibility of the services.

[1]This material can be used for non-commercial purposes under the CC BY-NC-SA 4.0 licence. Please contact the author concerning commercial usage.

Service Couplings with State and Function - in source or at run-time

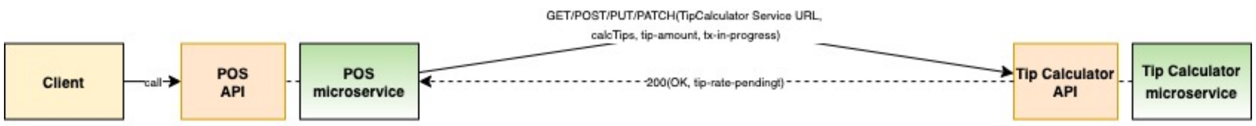

GET/POST/PUT/PATCH(TipCalculator Service URL, calcTips, tip-amount, tx-in-progress)
200(OK, tip-rate-pendingt)
Client
call
POS API
POS microservice
Tip Calculator API
Tip Calculator microservice

Figure 3. State Coupling in source or at run-time

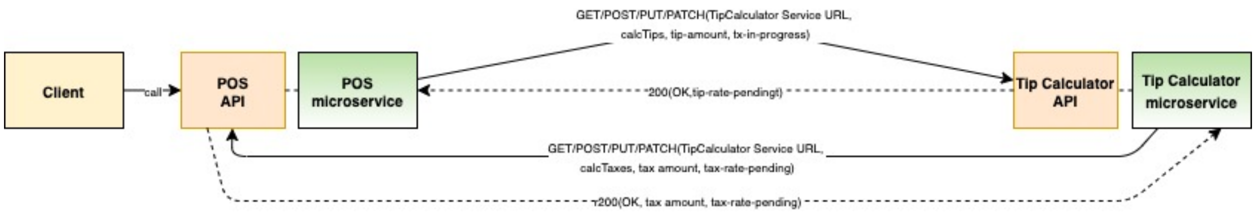

GET/POST/PUT/PATCH(TipCalculator Service URL, calcTips, tip-amount, tx-in-progress)
200(OK,tip-rate-pendingt)
GET/POST/PUT/PATCH(TipCalculator Service URL, calcTaxes, tax amount, tax-rate-pending)
200(OK, tax amount, tax-rate-pending)
Client
call
POS API
POS microservice
Tip Calculator API
Tip Calculator microservice

Figure 4. Bi-lateral State Coupling in source or at run-time

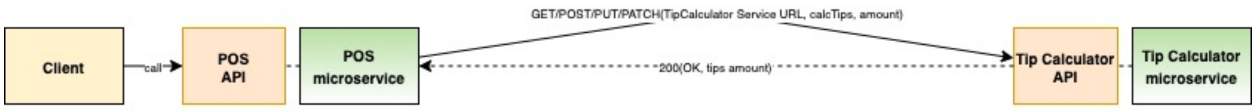

GET/POST/PUT/PATCH(TipCalculator Service URL, calcTips, amount)
200(OK, tips amount)
Client
call
POS API
POS microservice
Tip Calculator API
Tip Calculator microservice

Figure 5. Function Coupling in source or at run-time

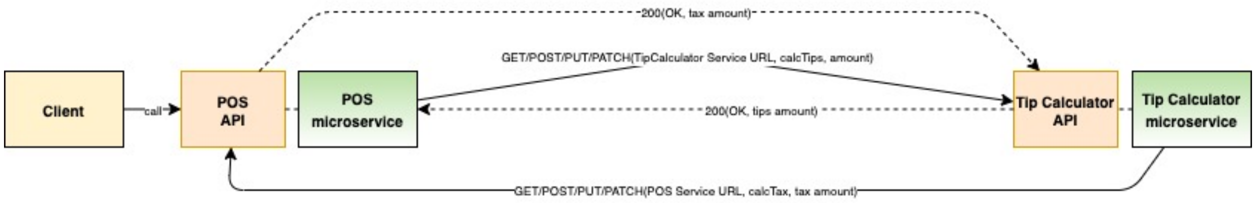

200(OK, tax amount)
GET/POST/PUT/PATCH(TipCalculator Service URL, calcTips, amount)
200(OK, tips amount)
GET/POST/PUT/PATCH(POS Service URL, calcTax, tax amount)
Client
call
POS API
POS microservice
Tip Calculator API
Tip Calculator microservice

Figure 6. Bi-lateral Function Coupling in source or at run-time

Another important dimension to service coupling is the *degree* of service coupling. To simplify the discussion, we define three distinct degrees of service coupling: *strong, weak, and loose*, that are inspired by Dynamic Coupling Measurement in OO software and the related works [4, 5]. In the case of *strong coupling,* as in Figures 3 and 4, a service consumer establishes control and data dependencies (*control and data couplings*) on both the behaviour and the state of the service provider in its source. Any change to the service provider would require changes to the service consumer. Likewise, if the provider also establishes control and data dependencies on the consumer in its source, both the service consumer and provider would be directly implementation-dependent on each other. It is said that they are *bilaterally strongly coupled*; they have achieved the highest degree of coupling. In another case of *strong coupling* as in Figures 5 and 6, a service consumer establishes control and/or data dependency on the behaviour but not the state of the service provider in its source or at run-time. Any change to the behaviour of the service provider would most likely require changes to the service consumer. Likewise, if the provider also establishes a control or data dependency on the consumer in its source or at run-time, both the service consumer and provider would be directly execution-dependent on each other. It is said that they are *bilaterally strongly coupled*.

In the case of *weak coupling,* as in Figures 7 and 8, a service consumer establishes a control dependency on the behaviour of the service provider through the explicit function references of the target service both in its source and at run-time, or vice versa. However, there is no dependency on the internal state. Their communication is usually message driven. Any change to the behaviour of the service provider may or may not require change to the service consumer. Finally, in the case of *loose coupling,* as in Figures 9 and 10, the service consumer and provider may have data dependency through message passing; however, neither the service consumer nor the service provider has any control dependency on each other. From another perspective, loose coupling is always indirect and involves an intermediary to decouple the brokered parties. Their inter-communication is message driven; they communicate using messages via an intermediary, such as a broker. Neither the consumer nor the provider needs to be aware of each other. Also, the broker may or may not have any dependencies on either party. Any change to the behaviour of the service provider would unlikely require change to the service consumer. As a result, loosely coupled services are easier to be reused and extended than other services with stronger couplings. In the following sections, we will demonstrate our methods of applying *loose service couplings* in our micro-messaging mesh pattern (MiMH) to achieve better reusability, extendibility, performance, and scalability than existing microservice architectures like service mesh does.

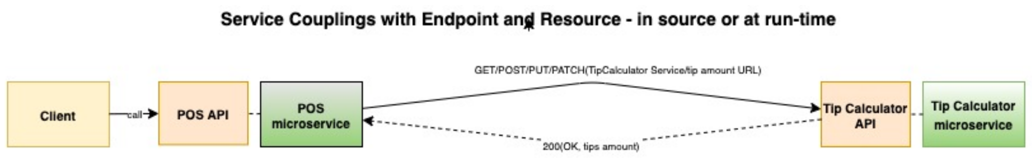

Service Couplings with Endpoint and Resource - in source or at run-time
Client
call
POS API
POS microservice
GET/POST/PUT/PATCH(TipCalculator Service/tip amount URL)
200(OK, tips amount)
Tip Calculator API
Tip Calculator microservice

Figure 7. Endpoint Coupling in source or at run-time

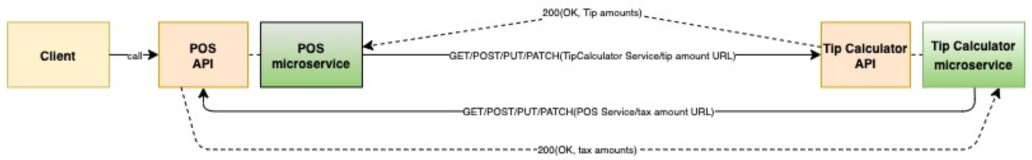

Client
call
POS API
POS microservice
200(OK, Tip amounts)
GET/POST/PUT/PATCH(TipCalculator Service/tip amount URL)
Tip Calculator API
Tip Calculator microservice
GET/POST/PUT/PATCH(POS Service/tax amount URL)
200(OK, tax amounts)

Figure 8. Bi-lateral Endpoint Coupling in source or at run-time

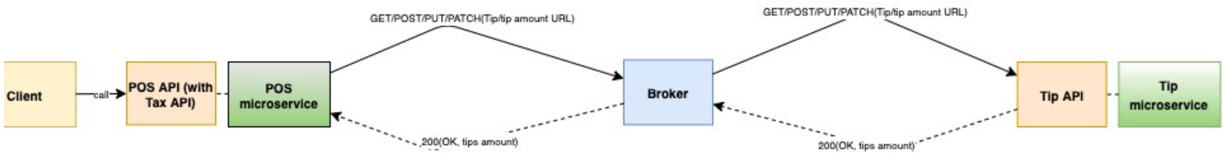


Figure 9. Resource Coupling in source or at run-time

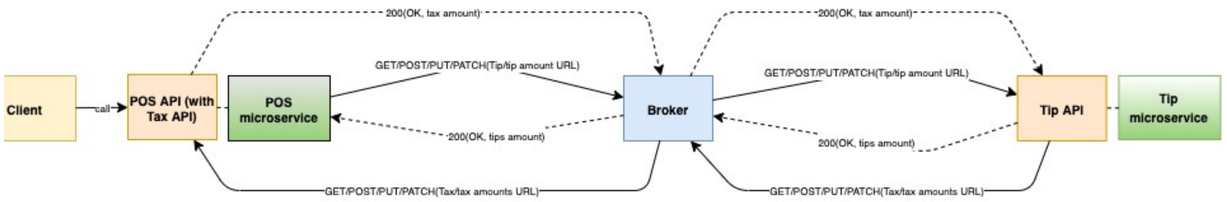


Figure 10. Bi-lateral Resource Coupling in source or at run-time

VI. An Introduction to Micro-Messaging Mesh (MiMH)

There are a number of problems in defining, designing, and building reusable, extendable, and scalable services using existing SOA patterns and frameworks. The existing SOA frameworks tend to be complex, require high overhead, and fail on reusability, and extendibility of the services. Most recently, the microservice pattern has emerged as solution for these problems. In particular, there is also a need for a flexible and loosely-coupled microservices architecture that not only minimizes integration effort, but also supports popular technology platforms and stacks without reinventing the wheel, especially in the cloud native environments. However, the microservices patterns in general have not lived up to its early promise of high performance, efficiency, resiliency, scalability, reusability, and extendibility [3, 6, 7]. In particular, service mesh has been proposed as one of the microservices patterns that provide high performance and scalability to microservices. However, service mesh uses a mesh communication model that is not necessarily efficient for many interaction patterns. Furthermore, a service mesh places too much emphasis and complexity on the sidecar proxies; the scope of sidecar proxy is not well-defined. Hence, service mesh has yet to fulfill its promises as well.

Being unsatisfied with the existing solutions and seeking a balance among all these concerns, we present a novel microservices pattern called *messaging or micro-messaging mesh (MiMH)* that extends the service mesh pattern. This concept was inspired by a hobby project of a team member during his undergraduate study in the early 1990s; it involved small pieces of software collaborating and using each other as intermediaries, with the ability to spread and scale out/in across his university's MTS Mainframe and Solaris systems; it was fun but not very rewarding.

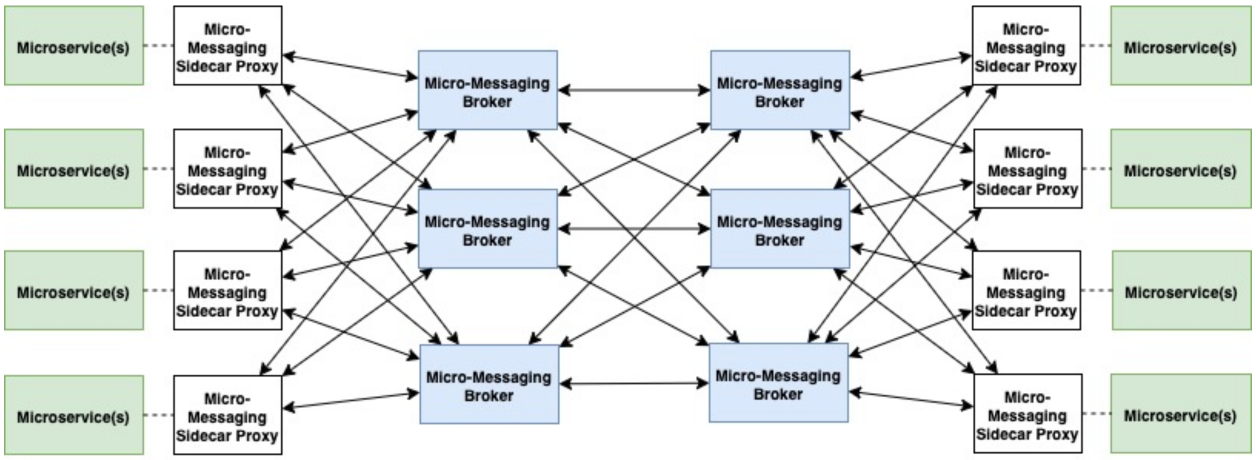


Figure 11. Overview of Micro-Messaging Mesh (MiMH)

With that said, the intent of this pattern is to improve loose couplings among microservices and enable separation of concerns or responsibilities. The motivation of this pattern is to provide additional loose coupling structure to the microservices while maintaining separation of concerns or responsibilities among intermediaries without reinventing the wheel. In addition, the service mesh pattern lacks specificity on the service couplings and mediation. Therefore, we apply the mediator pattern to the service mesh pattern, using intermediaries such as proxies or brokers to decouple the microservices as actors, and forming a composite pattern called *messaging mesh (MiMH)* to address these problems. In particular, this pattern is applicable to microservices that require messaging orchestration for their application use cases. It uses micro-messaging brokers for messaging orchestration, and keeps the sidecar proxies as lightweight proxies independent from messaging orchestration.

Figure 11 shows the structure of the *messaging or micro-messaging mesh pattern (MiMH)*. The participants of this patterns are: microservices, micro-messaging sidecar proxies, and micro-messaging brokers. They collaborate with each other to form a *micro-messaging mesh (MiMH)* that extends service mesh. A micro-messaging mesh allows microservices to communicate with each other via their micro-messaging sidecar proxy, which in turn communicates with its neighbouring micro-messaging brokers in a mesh topology. From one aspect, a MiMH can be viewed as a logical communication network of partial and/or full mesh models. It maintains loose service couplings by segmenting its partial mesh topology into three logical partitions, as in Figure 11; (i) between microservices and micro-messaging sidecar proxies, (ii) between micro-messaging sidecar proxies and the micro-messaging brokers, (iii) between micro-messaging brokers.

Unlike a service mesh that encourages a full mesh logical communication model [3], a MiMH has a partial mesh topology. While a full mesh topology appears to offer fast response time with direct point-to-point communication, all the nodes would need to possess equivalent communication capabilities. However, our assumption is that not all the nodes need to inherit all the capabilities in practice. Some nodes such as micro-messaging sidecar proxies or even micro-messaging brokers may be deployed in environments with limited computational resources and connectivity. In addition, a full mesh approach often not only increases the degree of service couplings but also burdens nodes with unnecessary structural and run-time overhead.

First, the logical communication partition (i) between microservices and micro-messaging sidecar proxies is a point-to-point model. A micro-messaging sidecar proxy provides the associated microservices with basic sidecar proxy capabilities including but not limited to: service discovery, telemetry, message routing, and/or message translation. It should be stateless and has very small footprint. It can also serve as a facade for the associated microservices. Secondly, the logical communication partition (ii) between micro-messaging sidecar proxies and micro-messaging brokers is a partial mesh. A micro-messaging broker provides the capabilities of orchestrating and aggregating messages among microservices via micro-messaging sidecar proxies. Unlike a typical message broker, a micro-messaging broker is light weight and should not be a generic message broker; it should be specialized to a set of message flow capabilities specific to an application or microservice domain. It may also support other messaging protocols that the microservices may not support. This enables a micro-messaging sidecar proxy to remain light weight and also off-load more complex messaging capabilities to the brokers. Thirdly, the logical communication partition (iii) between micro-messaging brokers could be a partial or full mesh. A message broker may coordinate and orchestrate message flows with other neighbouring brokers. This enables a micro-messaging broker to be optimized for specific capabilities, and also to apply the most effective message flow policies in distributing its messages to its neighbouring micro-messaging brokers.

The consequences of this pattern are enabling different combinations of categories, types, and degrees of service couplings among microservices using micro-messaging sidecar proxies and brokers. It allows the microservices to maintain a small footprint. It also allows the micro-messaging brokers to collaborate for supporting more complex messaging flows; and hence providing the flexibility of routing its messages to its neighbouring micro-messaging brokers in the most efficient manner. From another aspect, the micro-messaging brokers can also be viewed as an anti-pattern; they are not generic message brokers or mediators that emphasize on all-purpose message orchestration. They are light-weight messaging brokers specific to an application or microservice domain. They provide synchronous, asynchronous, and/or semi-synchronous messaging capabilities, and they can also orchestrate message by coordinating with each other. Therefore, they provide loosely coupled communication among microservices; and hence, improving their reusability and extendibility.

As a result, this pattern offers many advantages; it is platform, technology, and protocol agnostic. It not only minimizes integration effort, but also supports popular service oriented or microservices platforms and stacks without reinventing the wheel. This pattern enforces loose service couplings among microservices through the micro-message brokers by decomposing the API set into smaller loosely coupled sets that are separated by responsibility. This allows independent components to be modified without requiring changes on the upstream components; and hence improving the reusability, extendibility, performance, and scalability of all the participating components, especially in the cloud native environments. Additionally, this pattern enables high performance distributed systems and applications to be assembled with loosely coupled microservices using micro-messaging brokers. The micro-messaging brokers can coordinate with each and distribute messages among microservices in high availability (redundant) configuration; and hence provide resiliency and scalability to the microservices.

On the other hand, this pattern has a couple disadvantages; first, a technologist needs to put effort into assessing the interactions among the participating components (e.g. microservices, sidecar proxies, micro-messaging brokers) for achieving the optimal performance and service coupling. Secondly, a technologist needs to possess the technical skill in allocating the optimal responsibilities to the participating components. More specifically, a technologist needs to understand the types and degrees of the interactions and their associated attributes that have implications to the allocations of responsibilities to the components. With that said, the guiding principle of achieving the minimum response time is to minimize the number of intermediaries. However, this is usually insufficient; a better approach is to separate the responsibilities among micro-messaging brokers. The following section will describe a reference implementation using this approach.

VII. Reference Implementation and Proof of Concept For COVID-19

To validate our messaging or micro-messaging mesh (MiMH) pattern, we have developed a reference implementation of our micro-messaging mesh (MiMH) framework; using the micro-messaging pattern and our custom REST-based messaging called micro-messages for inter-component communication. Our goal is to develop a distributed framework using cost-effective computing resources including strictly open source tools under limited budget. It should support data processing for variable workloads and high volume of user queries comprising variable size message payloads at high performance targets. In particular, our objectives are to show that we can implement our MiMH framework using existing technology (e.g. kubernetes, RabbitMQ, NoSQL, Java, Golang, Python), and physical protocol stacks (e.g. HTTP, JSON-REST, gRPC, AMQP), with (i) reasonable time and effort, (ii) minimal infrastructure cost and footprint, and (iii) comparable response time and throughput to similar enterprise business applications that require much larger footprint.

In addition, we have selected the use cases of allowing users to query for the past and current exposure risk of COVID-19 as our proof-of-concept (PoC). More specifically, the PoC system comprises a web user application and a cloud native service platform that allows users to query for the current and past metrics relating to the exposure risk of COVID-19 using publicly available COVID-19 data sets, social activity statistics, and user geolocations. Most importantly, it also allows users to query for the forecast of exposure risk of COVID-19 by geolocation(s). The first version of the reference implementation includes constructing a pandemic data analytic and processing system based on the MiMH pattern. It evaluates the risk of exposure as specified in the user queries by analyzing publicly available COVID-19 data sets, social activity statistics, and user geolocations.

Furthermore, the POC objectives are: (i) it should require no more than one part-time technologist to develop and maintain the system on an on-going basis; (ii) its footprint must be small requiring 2 commodity budget servers; each server comprising 4 X 2.0 GHz cores; 8GB RAM; 500GB HDD; 100Mbps ethernet; (iii) all the servers use open source tools running on Linux; and (iv) the system should be deployed in active-active configuration for high availability. Additionally, we set our bar high for the performance targets of the system based on our theoretical estimation; it should be able to process at least 200 TPS or 200 user queries per second containing variable size payloads (<10KB); sustaining the peak load for 8 hours, and responding under 1.0 second with less than 1% errors. It should also be able to process ~1GB of incremental publicly source data daily with no errors, and storing an average of 500GB COVID-related data in the database without purging. That said, this paper provides an overview of the system architecture of the reference implementation and proof of concept; it is not intended for describing all the details. Instead, we will present our progress and more details about our technology in the future.

Figure 12 shows that the COVID-19 pandemic PoC system is built upon the reference implementation of our MiMH framework. It is made up of several microservices, micro-messaging sidecar proxies, micro-messaging brokers, and other functional components. Our framework can process large data sets and volume by using custom big data processing pipelines. It makes recommendations and forecasts by using a set of unsupervised learning algorithms in open source machine learning frameworks. More particularly, the COVID-19 data microservice queries for publicly available COVID-19 pandemic data sets from a number of reputable data sources such as WHO, Johns Hopkins University, CDC, and national health ministries. The data queries are performed in either real-time or batch mode. The Social Activity data microservice retrieves the most recent the statistics of social activities for different geographical areas from various public data sources. It also processes the geolocation data from the user mobile devices. The Exposure Risk microservice calculates and forecasts the COVID-19 exposure risk in each geolocation using the social activity statistics and the incident reports from the pandemic data sets.

More specifically, Figure 12 shows the system architecture of the pandemic system using the MiMH framework. It is a cloud native application deployed across multiple kubernetes clusters in a private cloud environment; consisting of a client tier, a front-end application service tier, a back-end broker tier, a back-end application service tier, and a persistent data tier. The client tier currently comprises a web user application implemented using javascript providing direct user interactions with the user. It communicates with the pandemic API gateway residing in the front-end application service tier. The front-end application service tier consists of a k8s front-end application service cluster comprising a pandemic API gateway, a pandemic service, and a micro-messaging sidecar proxy; each running on separate container and k8s node. The pandemic API gateway is implemented using an open source API gateway interfacing with a pandemic service implemented in Node.js, as well as the micro-messaging sidecar proxy. The pandemic service translates REST service requests originated from the web user application into back-end service requests via the micro-messaging sidecar proxy that is implemented using a lightweight open source HTTPS Proxy. The front-end tier communicates with the back-end microservice tier via the micro-messaging broker tier.

The back-end broker tier consists of a k8s micro-messaging broker cluster comprising three micro-messaging brokers; a micro-messaging broker that is implemented using Kafka; a micro-messaging broker that is implemented using Golang; a micro-messaging broker that is implemented using RabbitMQ; each running in a separate container and k8s node; all are deployed in a k8s cluster. The micro-messaging brokers provide message routing, forwarding, queuing, store-and-forward, translation, and transformation to its neighbouring micro-messaging components supporting these interaction scenarios: (i) among the pandemic service in the front-end application service tier and the microservices in the back-end application microservice tier, (ii) among the microservices in the back-end application microservice tier, (iii) among the micro-messaging brokers, the microservices, and the persistent data microservices in the persistent data microservice tier, and (iv) among the micro-messaging brokers in the micro-messaging broker tier.

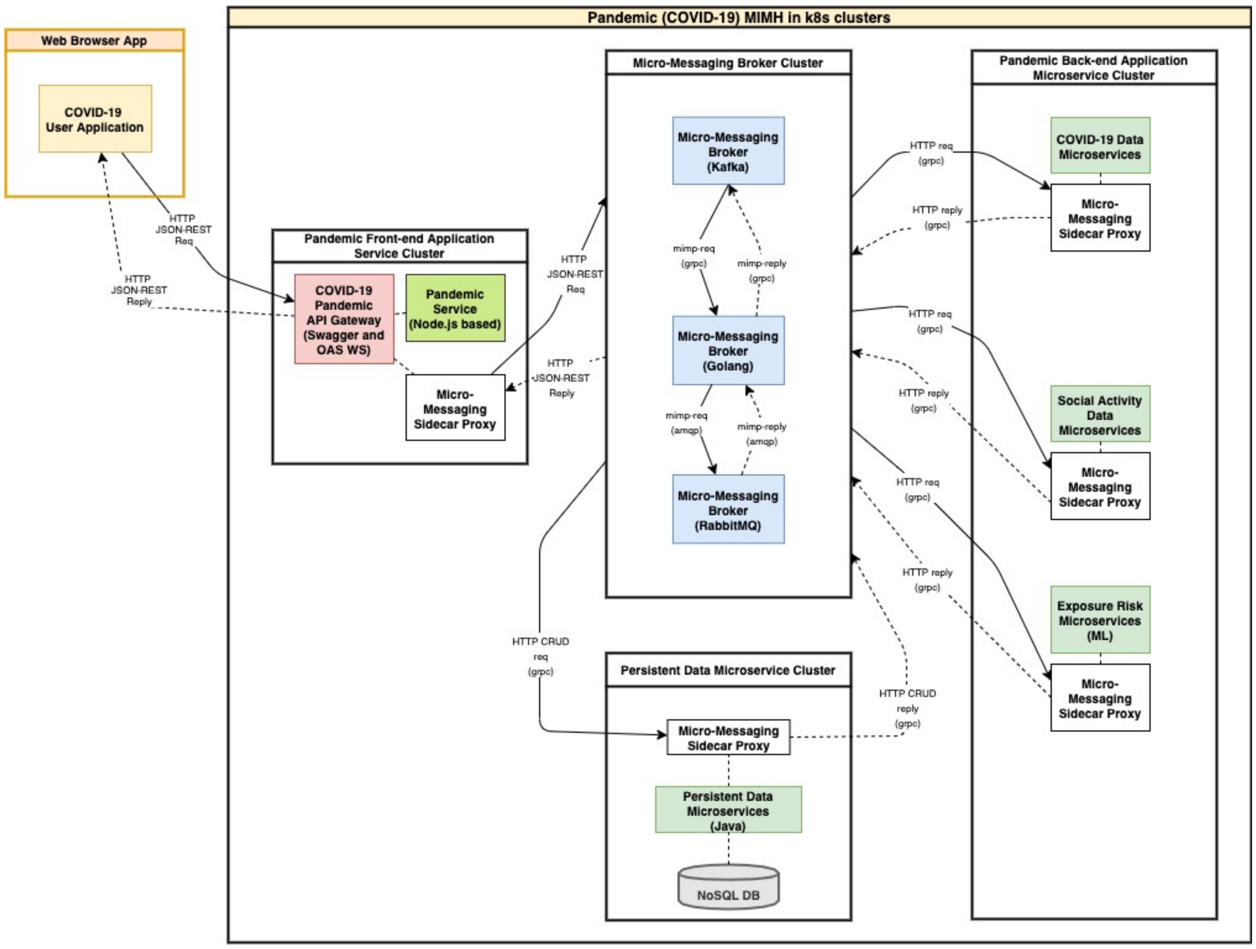
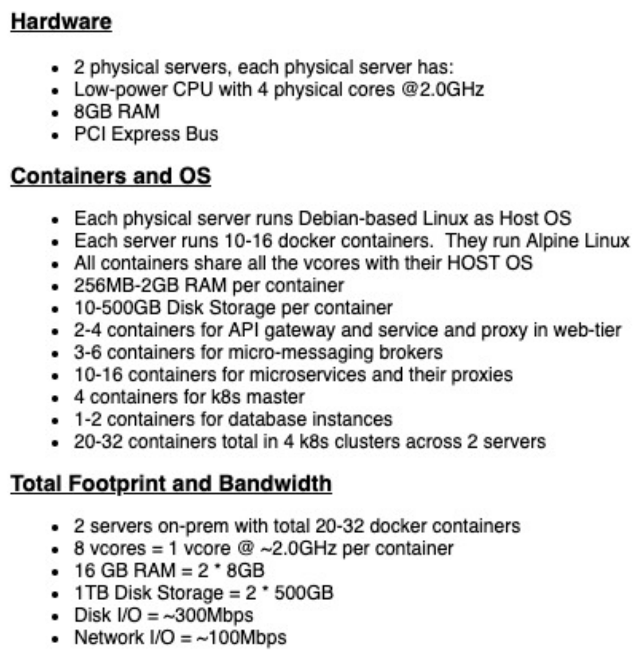

Pandemic (COVID-19) MIMH in k8s clusters
Web Browser App
COVID-19 User Application
HTTP JSON-REST Req
HTTP JSON-REST Reply
Pandemic Front-end Application Service Cluster
COVID-19 Pandemic API Gateway (Swagger and OAS WS)
Pandemic Service (Node.js based)
Micro-Messaging Sidecar Proxy
HTTP JSON-REST Req
HTTP JSON-REST Reply
Micro-Messaging Broker Cluster
Micro-Messaging Broker (Kafka)
mimp-req (grpc)
mimp-reply (grpc)
Micro-Messaging Broker (Golang)
mimp-req (amqp)
mimp-reply (amqp)
Micro-Messaging Broker (RabbitMQ)
Pandemic Back-end Application Microservice Cluster
HTTP req (grpc)
HTTP reply (grpc)
COVID-19 Data Microservices
Micro-Messaging Sidecar Proxy
Social Activity Data Microservices
Micro-Messaging Sidecar Proxy
Exposure Risk Microservices (ML)
Micro-Messaging Sidecar Proxy
HTTP CRUD req (grpc)
HTTP CRUD reply (grpc)
Persistent Data Microservice Cluster
Micro-Messaging Sidecar Proxy
Persistent Data Microservices (Java)
NoSQL DB

Figure 12. POC - Pandemic (COVID-19) PoC System with MiMH Reference Implementation

The back-end application service tier consists of a k8s back-end application microservice cluster comprising three groups of application data microservices; COVID-19 data microservices, Social Activity data microservices, and COVID Exposure Risk data microservice. The COVID-19 data microservices process COVID-19 public data sets. The Social Activity data microservices process general public social activity relating to their geolocations. The COVID-19 Exposure Risk data microservice analyze and forecast the exposure risk as specified in user queries. Furthermore, the persistent data service tier consists of a k8s persistent data microservice cluster comprising persistent data microservices, micro-messaging sidecar proxy, and the NoSQL database. This cluster allows the back-end microservices, micro-messaging brokers, and/or micro-messaging sidecar proxies to persist data in the lightweight NoSQL database instance(s).

To achieve optimal economy of scale and performance, we implement the external APIs using HTTPS JSON-REST API between the client tier and the front-end application service tier; the internal APIs using AMQP and gRPC over HTTPS. Our external APIs are synchronous while our internal APIs are a combination of synchronous and asynchronous messaging via micro-messaging brokers. All the REST APIs are implemented using our custom REST-based micro-messages that are optimized for low latency and message orchestration.

Consistent with our environmental friendly philosophy, our PoC system is built using ultra-low cost commodity hardware with low power consumption, as in Figure 13. That said, the commodity hardware provides below-average performance but is cost-effective and easy to replace. It is connected to the Internet with a download speed of ~50Mbps and an upload speed of ~10Mbps; however, the internal private network has a bandwidth of ~100Mbps. The persistent storage is a SATA HDD providing a below-average disk I/O of ~300Mbps. Total hardware cost for the two development servers listed in Figure 13 is less than $300 USD. Our framework uses exclusively open source software so there is no licensing cost.



Hardware

- 2 physical servers, each physical server has:
- Low-power CPU with 4 physical cores @2.0GHz
- 8GB RAM
- PCI Express Bus

Containers and OS

- Each physical server runs Debian-based Linux as Host OS
- Each server runs 10-16 docker containers. They run Alpine Linux
- All containers share all the vcores with their HOST OS
- 256MB-2GB RAM per container
- 10-500GB Disk Storage per container
- 2-4 containers for API gateway and service and proxy in web-tier
- 3-6 containers for micro-messaging brokers
- 10-16 containers for microservices and their proxies
- 4 containers for k8s master
- 1-2 containers for database instances
- 20-32 containers total in 4 k8s clusters across 2 servers

Total Footprint and Bandwidth

- 2 servers on-prem with total 20-32 docker containers
- 8 vcores = 1 vcore @ ~2.0GHz per container
- 16 GB RAM = 2 * 8GB
- 1TB Disk Storage = 2 * 500GB
- Disk I/O = ~300Mbps
- Network I/O = ~100Mbps

Figure 13. POC Sizing - Pandemic (COVID-19) PoC System with MiMH Reference Implementation

Furthermore, the basic deployment requiring minimum number of nodes or containers is a single stack with no redundancy. The higher-end deployment requiring almost twice the number of nodes or containers is an active-active configuration. Our HTTPS proxies serve as intelligent load balancers providing cost-effective and high performance load balancing using custom algorithms. Finally, the total build effort for the entire system including hardware, networking, and software requires ~200 person-hours up to this point. However, the system is built to be low maintenance, and hence, the on-going operating cost is expected to be minimal.

[1]This material can be used for non-commercial purposes under the CC BY-NC-SA 4.0 licence.
Please contact the author concerning commercial usage.

VIII. Conclusions and Future Work

We have completed the alpha version of our PoC system that ingests publicly available data from multiple data sources, aggregates the data sets, serves user requests, processes the data, and generates the forecasts. We have followed our micro-messaging mesh (MiMH) pattern and framework rigorously in designing and developing our sizeable codebase on top of open source tools. First, we identified our use cases early on and translated them into roles and responsibilities for individual components. Secondly, we evaluated the service couplings among all the components; resulting into effective component decomposition in an iterative manner. During our development iterations, we experienced no additional effort in our MiMH development even though we constantly needed to refactor our design and code; break code and components apart; create, modify and delete components; as well as rewire the components. Furthermore, we believe that our focus on service couplings and MiMH has improved our productivity with the Agile development method; it would not be possible with classical SOA or microservice approach which lacks integration with software engineering methods.

That said, we are still conducting the functional tests on our PoC system; resolving issues with data integration and quality, as well as verifying the accuracy of our forecasts. The preliminary results suggest that our MiMH framework performs efficiently across k8s clusters with a medium volume of selected test loads. Although we have only one part-time technologist developing and managing the sizeable codebase and configuration for a large number of system components, we do not have the urgent need of developing complex DevOps CI/CD and monitoring. We are slowly automating our system with DevOps CI/CD tools as we follow our mantra; *keep it simple stupid (KISS) theory.*

Our next step is to resolve the functional testing issues and improve our forecasts. We are incorporating new data sets, as well as additional machine learning algorithms and tools, for improving the accuracy of our predictions and supporting new use cases. We are planning to deploy our POC system to the public clouds (AWS, Azure, GCP), and making it available to the general public at a later time. In addition, we are planning to make our MiMH framework interoperating with other service mesh solutions [12-14]. Most importantly, we are improving the reusability, extensibility, and resiliency aspect of our MiMH framework with more advanced service decoupling methods. Based on the outcomes of our future experiments, we are also planning to make some parts of our MiMH technology available to the public in the near future.

Consistent with our philosophy, we strive to develop high-quality, cost-effective, and highly scalable solutions that also maximize the utilization of every computing resource and investment possible. Most importantly, our mission is to make our technology available to as many users and organizations as possible. We intend to communicate our progress and describe our technology in more details in the future. Finally, we envision many applications for our solution that can help our clients in simplifying their high-volume, mission-critical, and/or complex business systems and applications, especially across business domains, including but not limited to: Digital Commerce, Healthcare, Public Sector, and Financial Services.

IX. References

[1] R. Fielding, "Architectural Styles and the Design of Network-based Software Architectures", PhD thesis, MIT, 2000.
[2] J. Lewis, M. Fowler, "Microservices - a definition of this new architectural term", https://martinfowler.com/articles/microservices.html, 2014.
[3] IBM Redhat, "What's a service mesh", https://www.redhat.com/en/topics/microservices/what-is-a-service-mesh, 2020.
[4] E. Arisholm, L. Briand, A. Foyen, "Dynamic Coupling Measurement for Object-Oriented Software", 2004.
[5] E. Fregnan, Tobias Baum, "A Survey on Software Coupling Relations and Tools,", 2016.
[6] N. Dragoni, S. Giallorenzo, A. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, L. Safina, "Microservices: yesterday, today, and tomorrow", 2016.
[7] G. Hohpe, B. Woolf, "Enterprise Integration Patterns", 1st Edition, October 2003.
[8] K.Y. Chan, "Methods for Designing Internet Telephony Services with Fewer Feature Interactions", https://github.com/kenchan-canuck/ken-public-docs/blob/master/ken_msee_final_thesis_highres.pdf, MASc. Thesis in Electrical Engineering, University of Ottawa, May 2003.
[9] K. Y. Chan, G. v. Bochmann, "Methods for Designing SIP Services in SDL with Fewer Feature Interactions", pp.59-76, SDL 2003: System Design, 11th International SDL Forum, July 2003.
[10] K.Y. Chan, G. v. Bochmann, "Modeling IETF Session Initiation Protocol with SDL", pp.352-373, SDL 2003: System Design, 11th International SDL Forum, July 2003.
[11] T. Terauchi, A. Aiken, "Witnessing side-effects", Article No. 15, ACM Transactions on Programming Languages and Systems, May 2005.
[12] Google and Istio, "What is Anthos Service Mesh", https://cloud.google.com/service-mesh/docs/overview", 2020.
[13] Amazon Web Service, "AWS App Mesh", https://aws.amazon,com/app-mesh, 2020.
[14] Microsoft Azure, "Azure Service Fabric Mesh", https://azure.microsoft.com/en-ca/services/service-fabric, 2020.