Microservices: Technology

acarlstein.com/

Posted by Alejandro G. Carlstein Ramos Mejia on December 21, 2016 January 18, 2017 About Programming / Architecture / Microservices

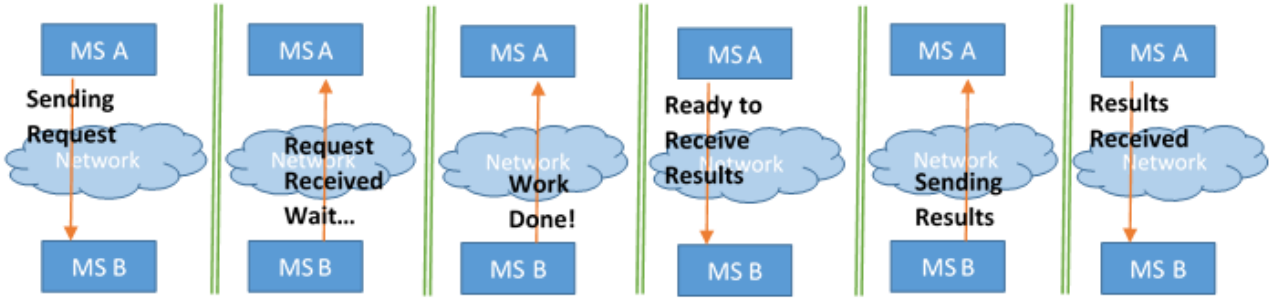
_Microservices: Design | Microservices: Brownfield: Approach

Communication

We have synchronous and asynchronous communication between the micro-services and/or clients.

Note: In real world applications, we use a combination of both.

Communication: Synchronous



This communication is based on making a request and waiting for a response.

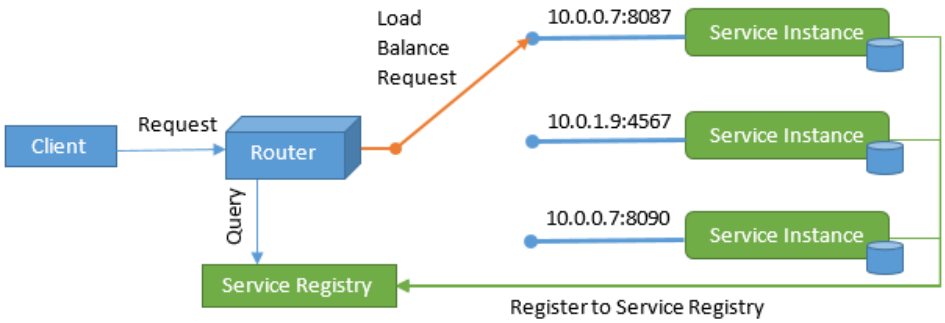
- Communication can happens between services, with clients and with external services.
- Remote Procedure Call
    - It makes it look like you are calling a method when in reality you are calling a remote method on a remote services.
    - Use Remote Procedure Call libraries
        - Help to Create distributed client-server programs
        - Provide functionality that do the work while shielding all the details regarding the network protocols and its communication protocols.

            HTTP communication protocol (it is firewall friendly)

            POST, PUT, GET, DELETE (Create, update, retrieve, and delete)
            - Create using POST
            - Update using PUT
            - Retrieve using GET
            - Delete using DELETE
            - Map to CRUD operations
    - Disadvantages:

It is sensitive to change (including at the server end); therefore, it breaks.
i.e.: Changing the method signatures

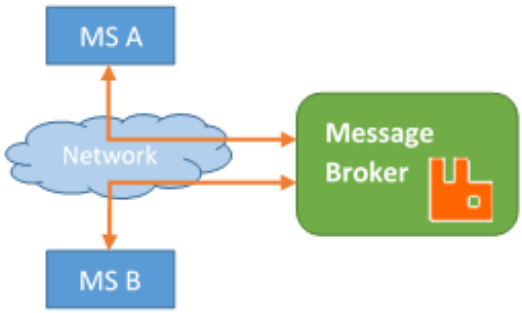
- Can use Request Response Synchronous
  REST
  - Open communication protocol
  - Natural decoupling
  - JSON/XML format
  - CRUD using HTTP verbs
  - Can access using endpoint URLs which are map to our entities.
  - Hypermedia As The Engine of Application States (HATEOAS)
    - Provides information to navigate the site's REST interface by using hypermedia links with the responses.
    - At different of SOA-based systems and WSDL-driven interfaces, its interface is dynamic which means that there is no fixed specifications that may be staged somewhere else on the site.
- Synchronous disadvantages:
  - Communication only available when both parties are available.
  - One micro-service must wait for the other microservice to response before it can carry on.
  - If one micro-service slow down, the entire transaction slow down.
  - It makes it speed dependent of the network speed.
    This would make responses from the micro-services to arrive slowly.
  - Micro-services must know the location of that service (host and port).
    Note: This issue can be solved by implementing a service registration and discovery pattern.



Communication: Asynchronous

Asynchronous communications means event-based communications. Instead of having our services connecting directly to another service to carry out a task, the service creates an event and micro-services that can carry on that task out will automatically pick that event up.

- There is no need for a client and server connection. Client and server are decoupled.
- A message queuing protocol is used where the events created by a micro-service (the publisher) will be send as messages, and the service which carries out the tasks in response to that event (the subscriber).

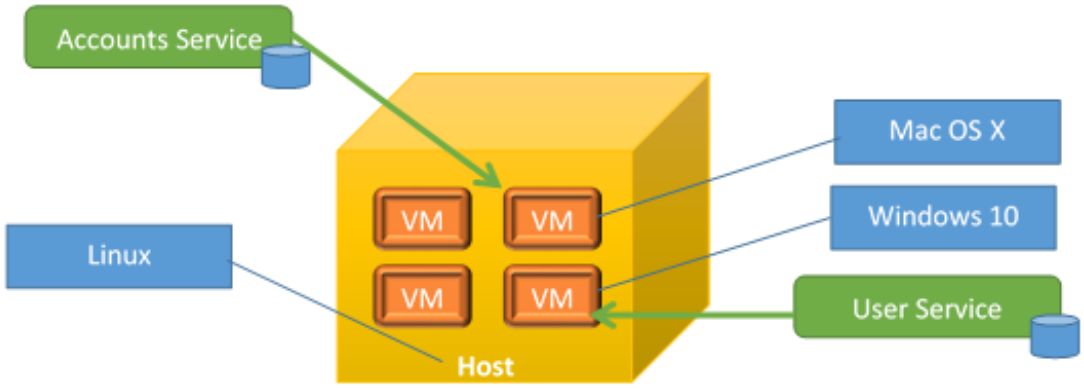


- All messages from publishers are stored and buffered by a message broker (i.e. RabbitMQ, Microsoft Message Queuing, ATOM) until a subscriber picks the message up.
    - The publisher only knows of the message broker.
    - Publisher and subscriber are decoupled; therefore, they don't need to know each other's location.
    - Note: ATOM use HTTP to propagate events.
- This communication protocol is perfect for micro-services architecture because we can swap different version of a micro-service without having to affect other micro-services.

    This is because they only need to know about the message broker and communicate using a message queuing protocol.
- You can have multiple subscribers acting on the same message.
- Multiple vendors provide message queuing protocols.
- Asynchronous communication disadvantages:
    - It's complicated when using distributed transactions.
    - The system relies on a message broker which is one point of failure for the whole system.
    - Visibility of transactions can be a problem.
    - The transactions are not completing in a synchronous way.
    - To manage messaging queue, you are required to learn new tools and techniques.
    - We need to make sure to handled property the messaging queue.
    - The performance of the system may be affected if you have queuing issues.

Virtualization

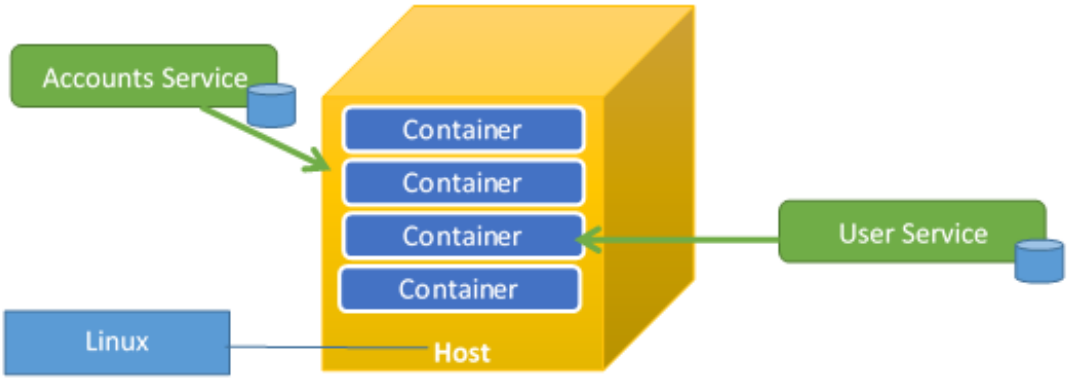
One way to host micro-services is virtualization which means that we are going to use virtual machines as host in order to run our micro-service



- There is no need to run our micro-services on physical machines. We can run a self-contained machine with our micro-service in it.
    - Spin a virtual machine
    - Install micro-service
    - Done
- Virtual machines are entire instances of operating systems (plus any extra software required).
- Hardware is simulated in software.
- One physical machine can run multiple virtual machines.
    Each virtual machine runs as they are physical machines themselves.
- Virtual machines can be cloned.
    When you have a virtual machine ready with your micro-service, you can clone several copies of that virtual machine.
- Virtual machines are the foundation of close platforms.
    - You can subscribe to cloud platform services known as “platform as a service” (PAAS).
    - You can run your whole micro-service in the cloud. i.e. Microsoft Azure platform, Amazon web services,
- As your demand increases, you can create multiple virtual machines with your complete architecture.
- There are platforms that allow you to create your own cloud.
- They take time to be set up, as well as time to load, and resources.
- You can take snapshot of a virtual machine. Such snapshot can be restore later.
- Current virtualization technology is standardized and very mature.

Containers

Containers are a different kind of virtualization. While they do not run an operative system within the container, they run the minimal amount of requirements needed for your micro-service to run.



- It allow to isolating services from each other.
- It is a good practice to run only one service, one micro-service within a container.
- They use less resources than a virtual machine.
- You can have more containers on a host machine than virtual machines.
- They are faster than virtual machines.
- They boot faster than virtual machines.
- Since containers are lightweight and fast, they are faster to create new instances.
- Clod platform support for them is growing. i.e. Microsoft Azure Platform, Amazon Web Services.
- Currently only supported on Linux but soon other operating systems will join.
- They are not quite yet establish and standardized as virtual machines.
- Quite complex to set up.
- Docker, Rocker and Glassware are examples of containers.

Hosting Platforms

Self-hosting can be an option you may pursue. While, cloud services allow you to control the whole project via a portal in a simpler way (and there is no need for physical machines or specialized staff), you may want to implement your own cloud service using your own IT infrastructure. This means that you will be implementing your own virtualization platform or implementing your own containers.

When implementing your own cloud have the following as consideration:

- Think in long-term maintenance of your could platform.
- Think about the need for technicians who will be supporting your cloud platform.
- Train your existent staff.
- Reserve or expand space for extra servers.
- Scaling will not be immediate (as buying an external service) due purchases of new machines and configuration required.

Registration and Discovery

When we connect to a micro-service we need to find out:

- Where the micro-service is.
- What host the micro-service is running on.
- What port it is listening on.
- What version of the micro-service is running.

One way to store this information (plus making such data available) is having (or implementing) a service registry database. Therefore, when we implement a new micro-service or we implement a new instance of a micro-service, we make sure our micro-service register itself on startup in this database. This is called register on startup.

In addition, we can make our system smart enough to recognize when a micro-service stop responding. In this way, the system deregister that instance of that micro-service from the service registry database. This is called deregister service on failure.

By doing these steps, new client requests will not connect to micro-services instances that are not available and/or experiencing problems.

The way our micro-services will register themselves in the service registry database is by:

- Option 1: Our micro-services register themselves to the registry database on startup.
- Option 2: (Third Party) software that detects new instances of micro-services and register them to the registry database.

The way our client learn about all the locations of our micro-services and micro-services instances is by:

- Option 1: Directly connecting to the service registry database in order to obtain the location of specific instance of our micro-services. This is called Client-side discovery.
- Option 2: Connecting to a gateway (or load balancer) which retrieves the location from the service registry database. This is called Server-side discovery.

Monitoring Technology for Observable Micro-Services

At our disposition we have monitoring centralised tools such as Nagios, PRTG, Sensu, Zabbix and mist.io; as well as other components such as load balancers. Also, there are online services such as New Relic, Uptime cloud monitoring, Ruxit, Site 24×7, which allows you to look into your system and monitor it.

Key features:

- Metrics across servers: Gather metrics across servers then aggregate and visualize data.
- Automatic and minimal configuration for new instances of our micro-services.
- Tools to allow us to use client libraries to send metrics.
- Tools to send test transactions, as well as to monitor those test transactions.

- Tools that send alerts when something being monitored is failing.
- Tools to monitor network and network transactions.
- Monitoring real time.

Note: We need to make sure that monitoring is standardized across the whole system. This means that we need to ensure all our hosts, virtual machines and/or containers are pre-configured with all the requirements for monitoring.

Logging Technology for Observable Micro-Services

We need a portal for centralised logging data so we can deal with the logging of specific events within our system. All logs get push into these centralised logging tools which store the data into a central database. Later, our portal should allows us to query and find patterns within this logging data. For example, we can use a Correlation ID and Transaction ID to verify the journey of a process (and/or transaction) through all our micro-services in our system. In this way, we can query such IDs within our centralised logging tool.

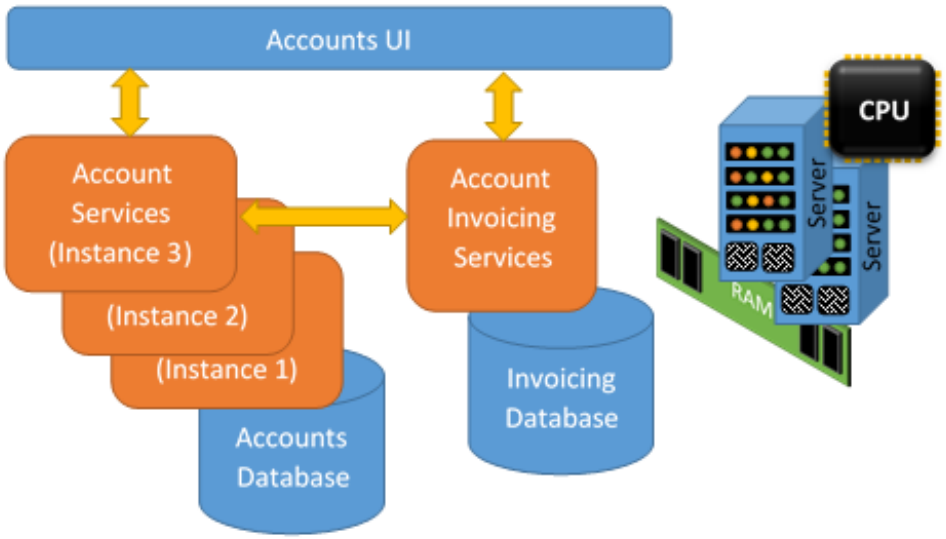
Remember that in our micro-services, we will need to implement a client logging library for the purpose of sending the logs into our centralised logging tool. An example is Serilog, Splunk, LogStash, Fluentd, Logmatic.io, and Graylog.

Key features:

- Support structured logging.
- Query logs.
- Minimal useful information.
- Logging across servers; therefore, accept data from multiple servers.
- Automatic or minimal in terms of configuration.
- It must allow to log a Correlation and/or Context ID.
- Allow for standardization of our logging.
    - We can use a template for our client libraries.
    - Ensure that the format of the log must be an standard across the system.

Scaling

When the performance requirements for our micro-service increases, which makes our micro-service performance to degrades, we must increase the number of instances of our micro-service and/or increase our resources.

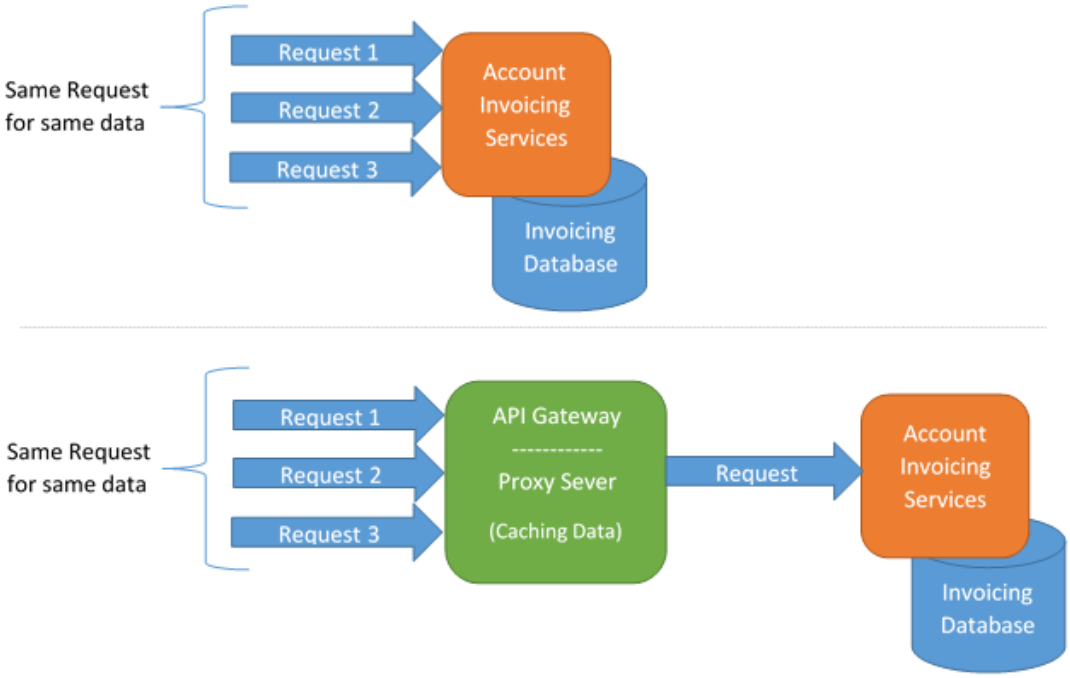


- Create multiple instances of the micro-service
- Add extra resources such as:
    - Increase the number of CPU calls available.
    - Increase the number of memory available.
    - Increase the amount disk space.
    - Increase the amount of bandwidth available.
    - Increase number of virtualization and/or containers.
    - Increase number of host servers.
- Amount of instances and/or resources available might be automated or based on-demand (manually).
    - Some of the on-demand resources may need to be increased in the hardware manually.
    - PAAS auto-scaling options.
- Load balancers

  API Gateway
- Only scale up when
    - There is performance issues.
    - The monitoring data shows that there are performance issues.
    - The capacity planning shows that you will have performance issues.

Data Caching

The performance of your micro-services architecture system can be improved by implementing caching of data. This is done by detecting multiple calls towards the same thing. So, instead of honoring each request individually, we can retrieve the data and use it over and over again while the request received is the same. In other words, we retrieve the data and keep it alive in order to satisfy all the other requests; therefore, improving performance by reducing the client calls to that specific service, service calls to a database, and/or service to service calls.



The API Gateway level or the proxy server level are the ideal place to do caching since it became “invisible”. This will reduce the number of calls to your micro-service (and database), plus it will reduce the amount of traffic within your own network.

Another way to do caching is at the client side by downloading most of the data they need to work. In that way, only when it is needed that a call is done to your system.

Caching can also be done at the service level. In that way when a call done to a micro-service (lets call it service A) equals the call of another micro-service (lets call it service B), then the first micro-service (A) can take care of both requests using the same data.

Our caching system must be simple to set up, managed and easy to implement.

When working with caching, we must be aware of data leaks. We don’t wish to present the wrong data to the wrong call.

API Gateways

API Gateways are the central point into the system. All the client applications must access via our API Gateway. We can use our API Gateway to improve performance by applying load balancing functionality, as well as caching functionality. The fact that our API Gateway is the central point of access, help us to simplify the implementation of caching and load balancing. For the rest of the system, the API Gateways becomes “invincible” since they don’t need to know how the load balancing and caching works.

- API Gateways provide one interface for many services.
- They take care of the load balancing and caching.
- It allows the scaling of micro-services without the client noticing.
- It allows to move micro-services around in terms of location without the client

noticing.
    It helps with dynamic location of services.
- The API Gateway can be configured to route specific calls to specific instances of a micro-service.
- The API Gateway can be configured to look up for the location of micro-services in the service registry database in which all micro-services register.
    This helps with load balancing.
- The API Gateway provide a level of security for the overall system.
    - We can provide authentication and authorization.
    - Central security versus service security level.
        - We can have a dedicated service for authentication and authorization working in the background.
        - We can have a service dedicated to the security for authentication and authorization.

Automation Tools

When making a decision for automation tools such as continuous integration, see if you can find the following features:

- The tool should be cross platform.
    The tool must have the correct cross-platform builders.
- The tool should integrate with the chosen source control system (such as Git, SVN, Perforce).
    It should detect a code change and trigger off the build.
- The tool should be able to send notifications.
- The ability to map a micro-service to a CI build.
    - Code change to a specific micro-service will only trigger the build for that specific micro-service.
    - We can place our micro-service in a specific place, helping us with the continuous deployment.
- Build and test should run quicker.
- There should be quick feedback.
- Separate code repository for service.
    We prevent two or more micro-services to change accidentally at the same time.
- We can configure the CI build to test the database for changes.
    This allows us to ensure that the micro-service and the database are upgraded.
- Note: Avoid having one CI build for all services since you will loose all the advantages previously described.
    - Each micro-service should have its own CI build.
    - Reminder: All micro-services should be independently changeable and deployable; therefore, we must have one CI build for each micro-service.

For tools such as continuous deployment, we wish.

- The tool should be cross platform and support multiple environments.
- Central control panel.
- Easy to add deployment targets to deliver our software.
- Allow for scripting when just copying files across in not enough.
- Allow for additional ad-hoc tasks.
- Build statuses highlighting things such as failing tests and such.
- Integration with CI tool.
- This tool can released to a cloud-based hosted system.
- Support for PAAS