Microservices

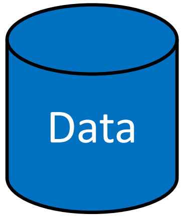
SAURABH SRIVASTAVA

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

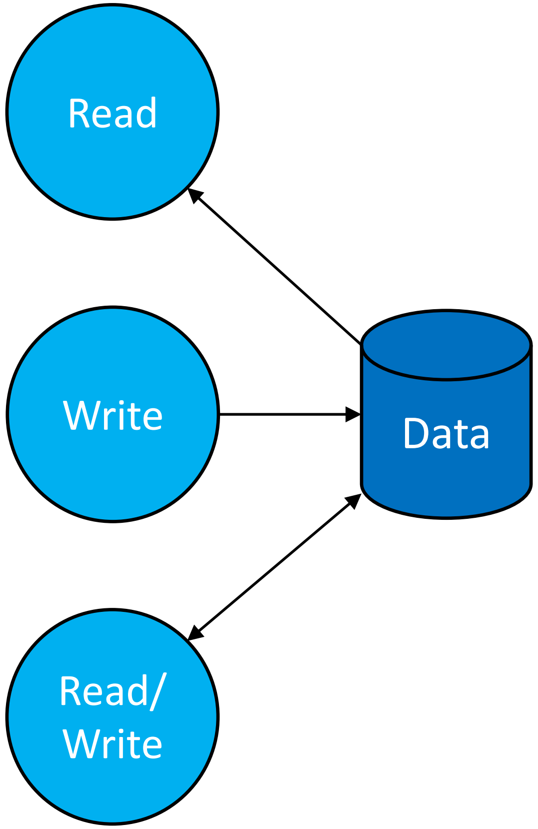
IIT KANPUR

How do we build Systems ??

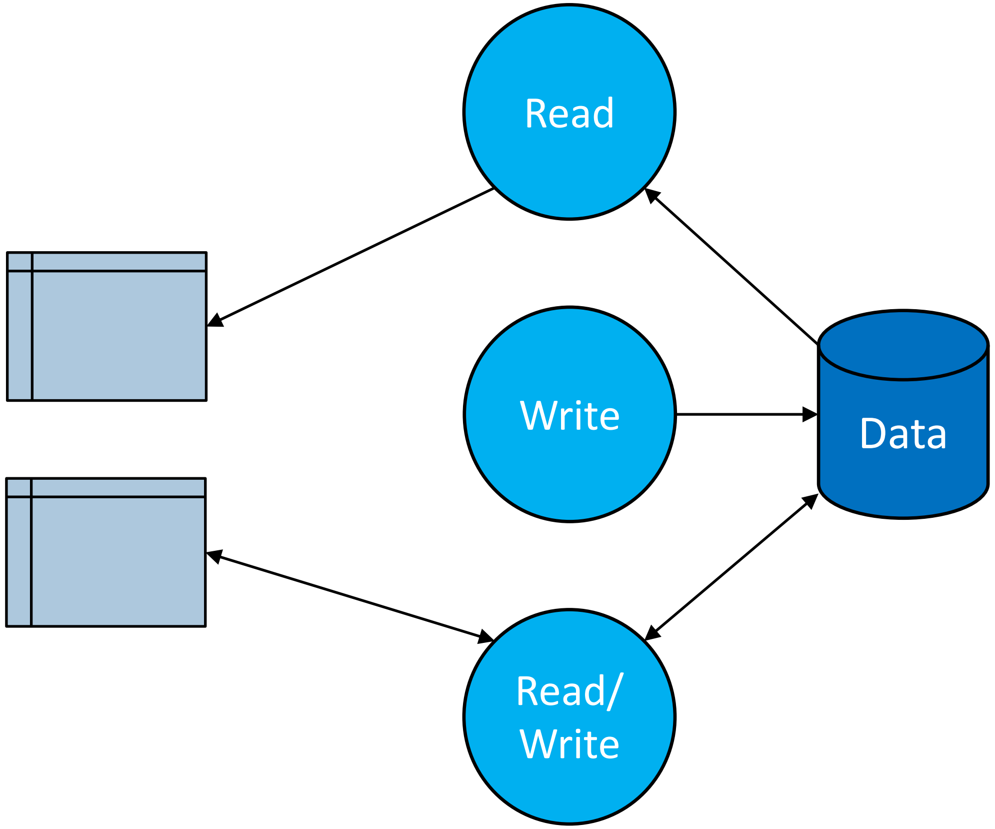
We Have Data

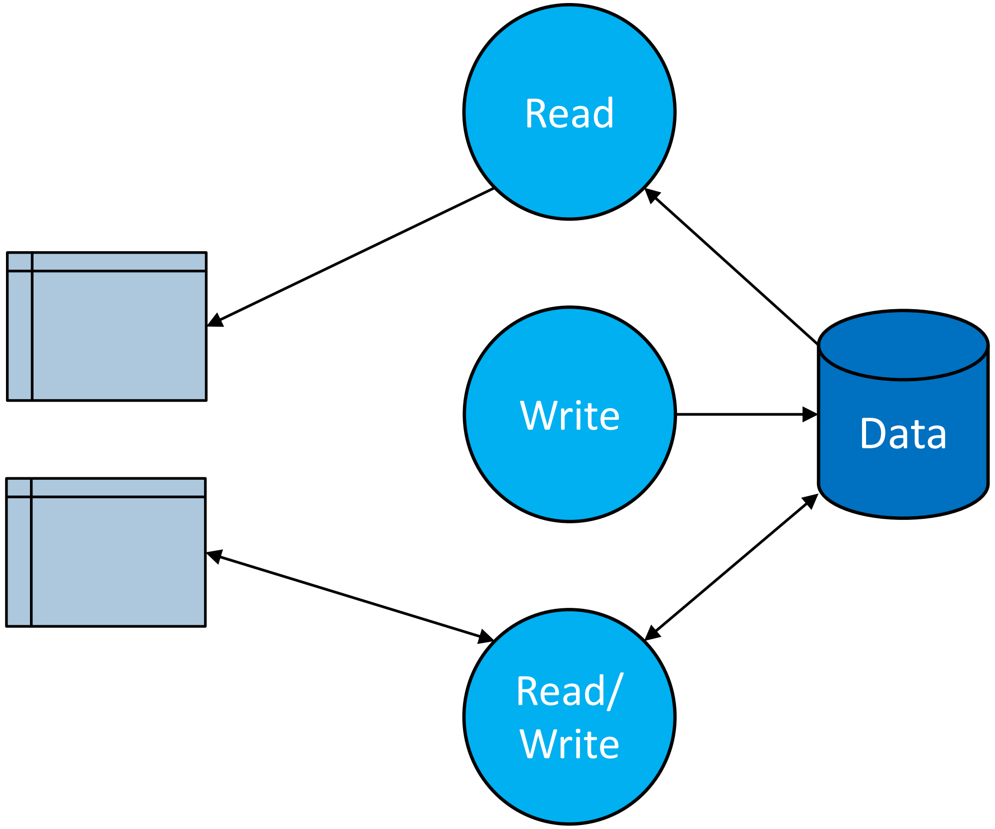


We write code
to access that data



We then provide interfaces to view or edit this data

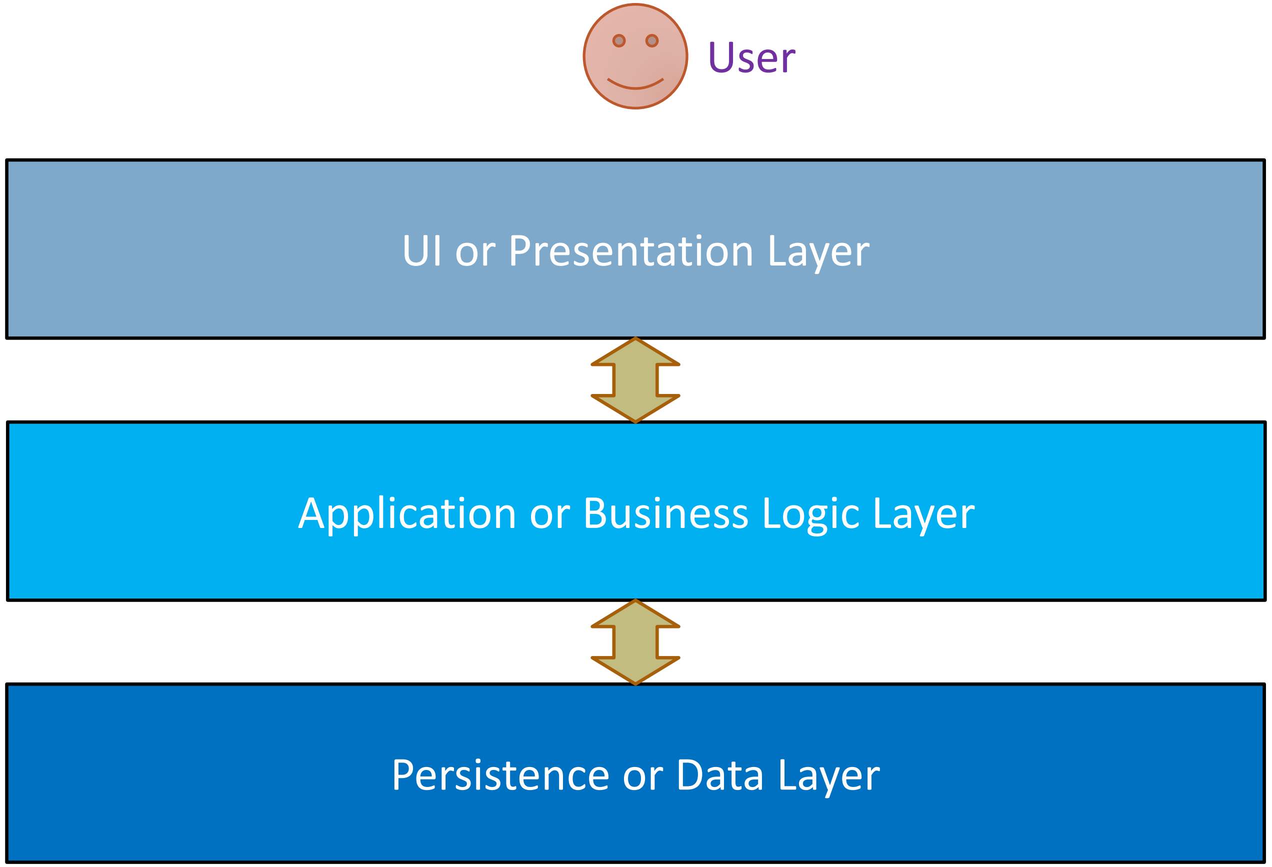




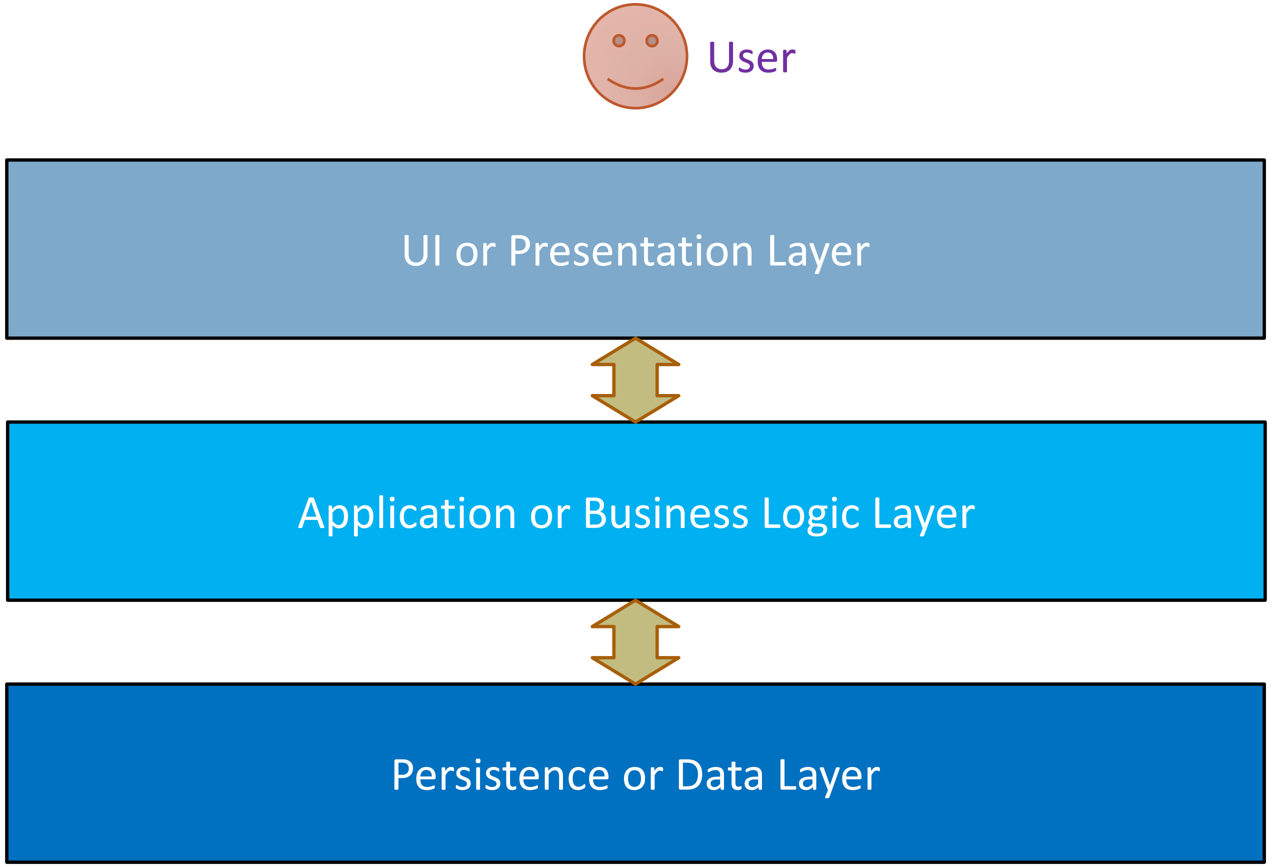
This is a common way to build applications, since very long

The methodology is based on the *3-Tier Architecture*

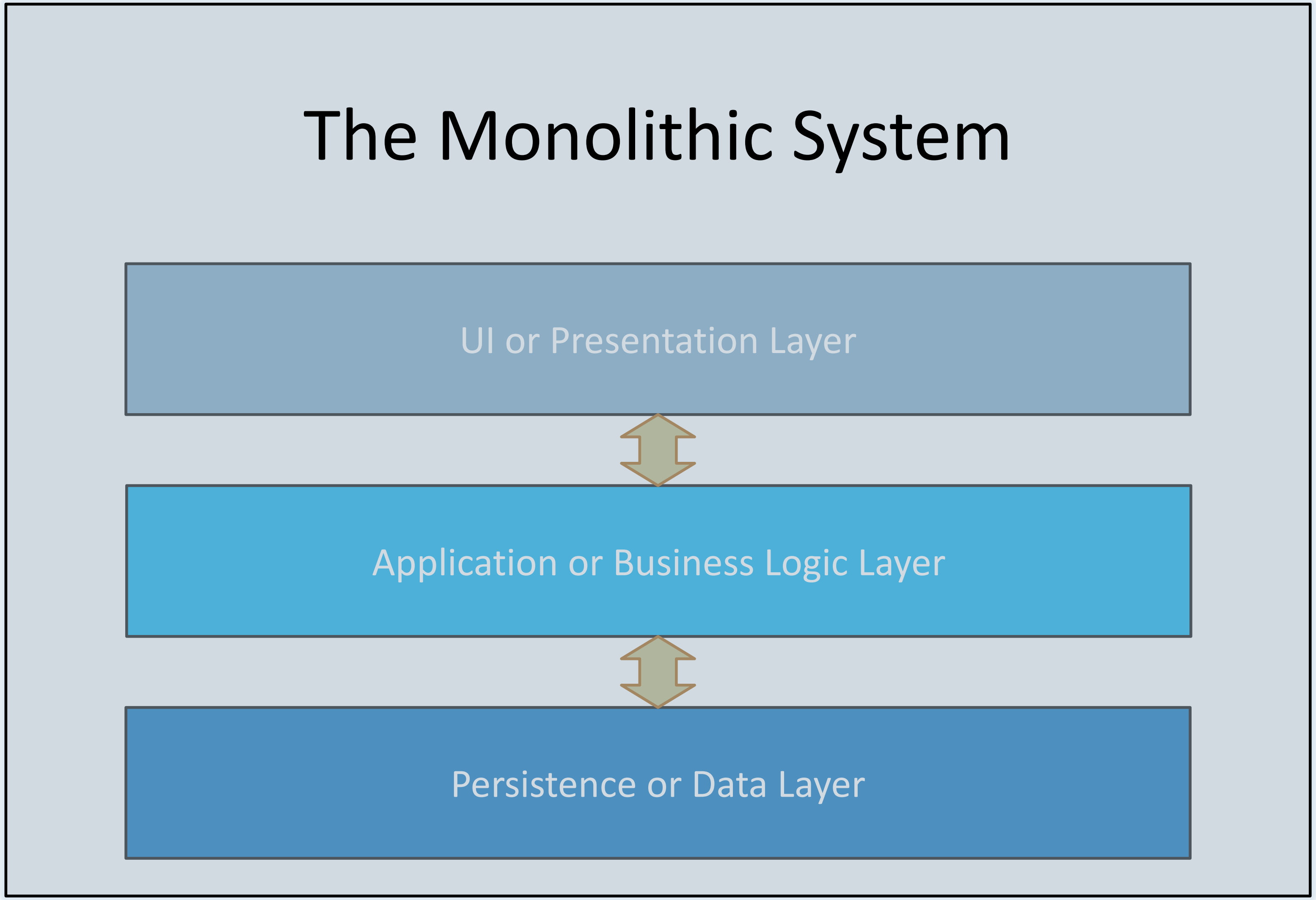
The methodology is based on the *3-Tier Architecture*


User
UI or Presentation Layer
Application or Business Logic Layer
Persistence or Data Layer

The methodology is based on the *3-Tier Architecture*


User
UI or Presentation Layer
Application or Business Logic Layer
Persistence or Data Layer

Coined around 2 decades ago !!



The Monolithic System

UI or Presentation Layer

Application or Business Logic Layer

Persistence or Data Layer

Do we need a change in how we build systems?

“If it ain't broke, don't fix it.”

Bret Lance ‘1977

Do the Enterprises care to change if everything is well?

Do the Enterprises care to change if everything is well?

Not really !!

Do the Enterprises care to change if everything is well?

Not really !!

So is everything well?

Do the Enterprises care to change if everything is well?

Not really !!

So is everything well?

Not really !!

Do the Enterprises care to change if everything is well?

Not really !!

So is everything well?

Not really !!

So what’s wrong with current architectures?

Time to Market !!

A typical enterprise software sees 2-6 releases a year

- That is about 2 months to release one version, at best !

New Features must be thoroughly tested before they can go live

- Unit Tests, Integration Tests, Regression Tests, User Acceptance Tests
- The testing itself may take a month or more to complete

Contrast that to applications on the web

- Facebook makes changes to the live code on a daily basis [1]
- Even multiple releases a day, is fairly common

Enterprises would love to decrease the time it takes to release new versions

- The current architecture may involve rebuilding a lot of things – may be everything – for a small change
- Even if it doesn’t the coupling between the components may mandate regression testing

Operating on scale ??

Internet is an utter chaos

- So much heterogeneity – from protocols to data formats to unreliable networks
- Yet some internet applications can somehow make it work, that too, on a large scale

Enterprise Solutions may need to go online (or may be already online)

- What if our systems are to be accessed by customers *over the internet* ?
- Can our *monoliths* handle the scale of internet?

Scaling vertically is an option, or is it?

- We can add more hardware to our servers, and they can handle more load (vertical scaling)
- But what if the application is *inherently incapable* to use new cores (no parallelism) ?

We can run multiple instances of the application

- Some parts of the application may be rarely used, but had to be replicated any way

Can’t we do better?

How about if we *split the monolith* ?

How about if we *split the monolith* ?

Instead of one system, have a *system of systems*

How about if we *split the monolith* ?

Instead of one system, have a *system of systems*

Each smaller system, is a system on it's own

How about if we *split the monolith* ?

Instead of one system, have a *system of systems*

Each smaller system, is a system on it’s own

It keeps it’s own data

How about if we *split the monolith* ?

Instead of one system, have a *system of systems*

Each smaller system, is a system on it's own

It keeps it's own data

Probably has it's own UI too

How about if we *split the monolith* ?

Instead of one system, have a *system of systems*

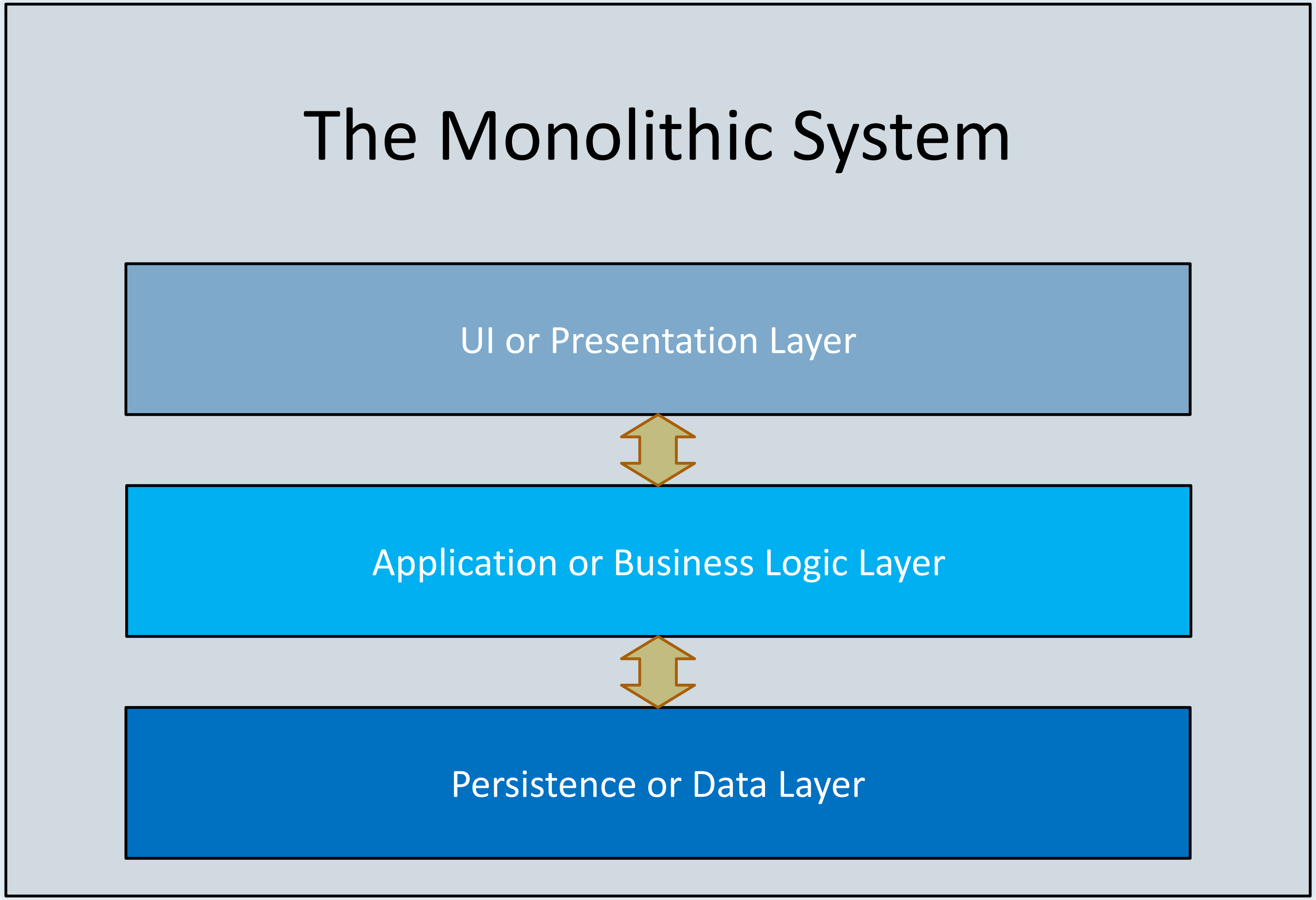
Each smaller system, is a system on it’s own

It keeps it’s own data

Probably has it’s own UI too

Exposes an interface for others to use

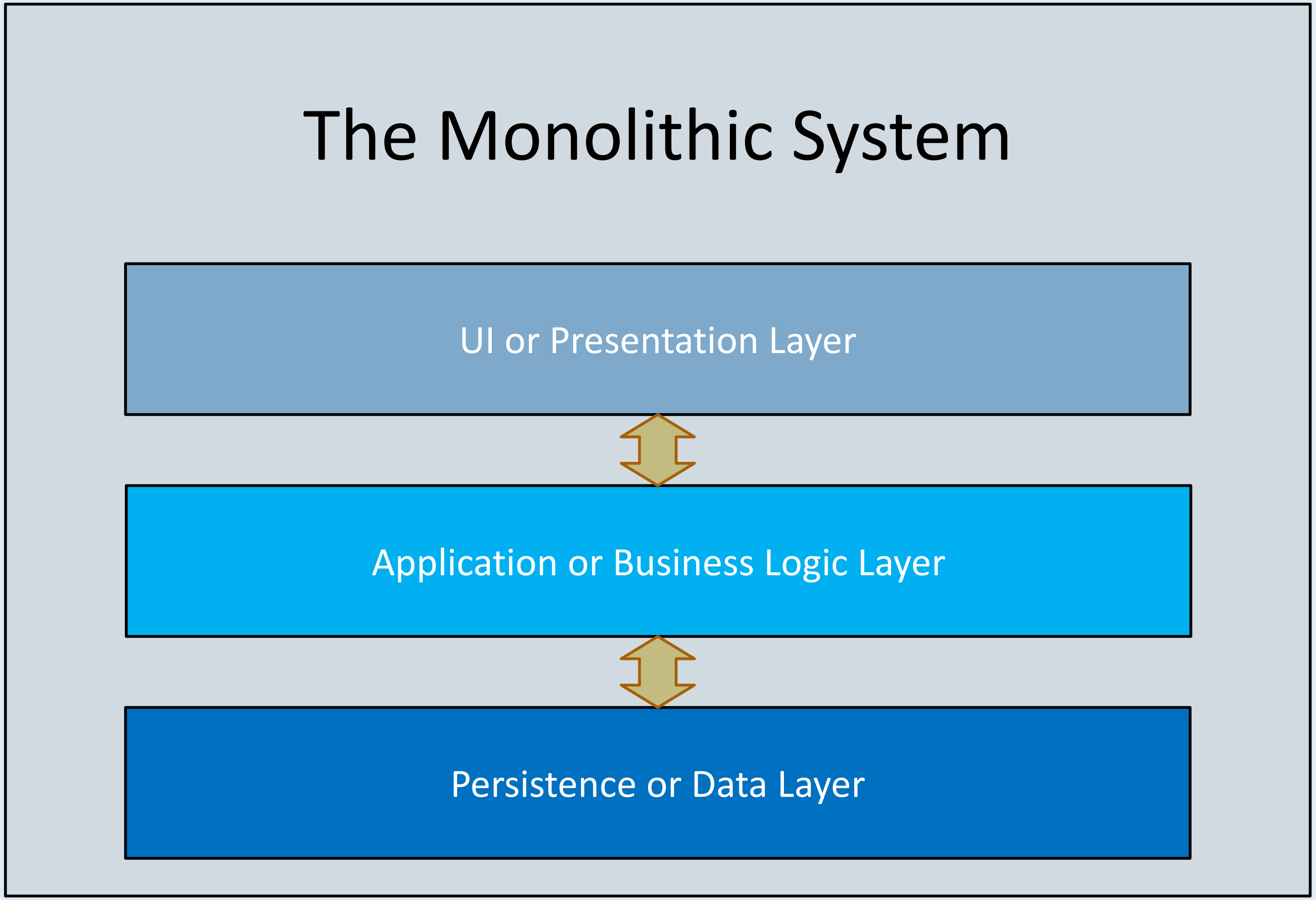
The Monolithic System


The Monolithic System
UI or Presentation Layer
Application or Business Logic Layer
Persistence or Data Layer

Right now
we are cutting it
horizontally

“Organizations which design systems ... are constrained to produce designs which are copies of the communication structures of these organizations.”

Conway’s Law ‘1968



The Monolithic System

UI or Presentation Layer

Application or Business Logic Layer

Persistence or Data Layer

We see it from a technical perspective

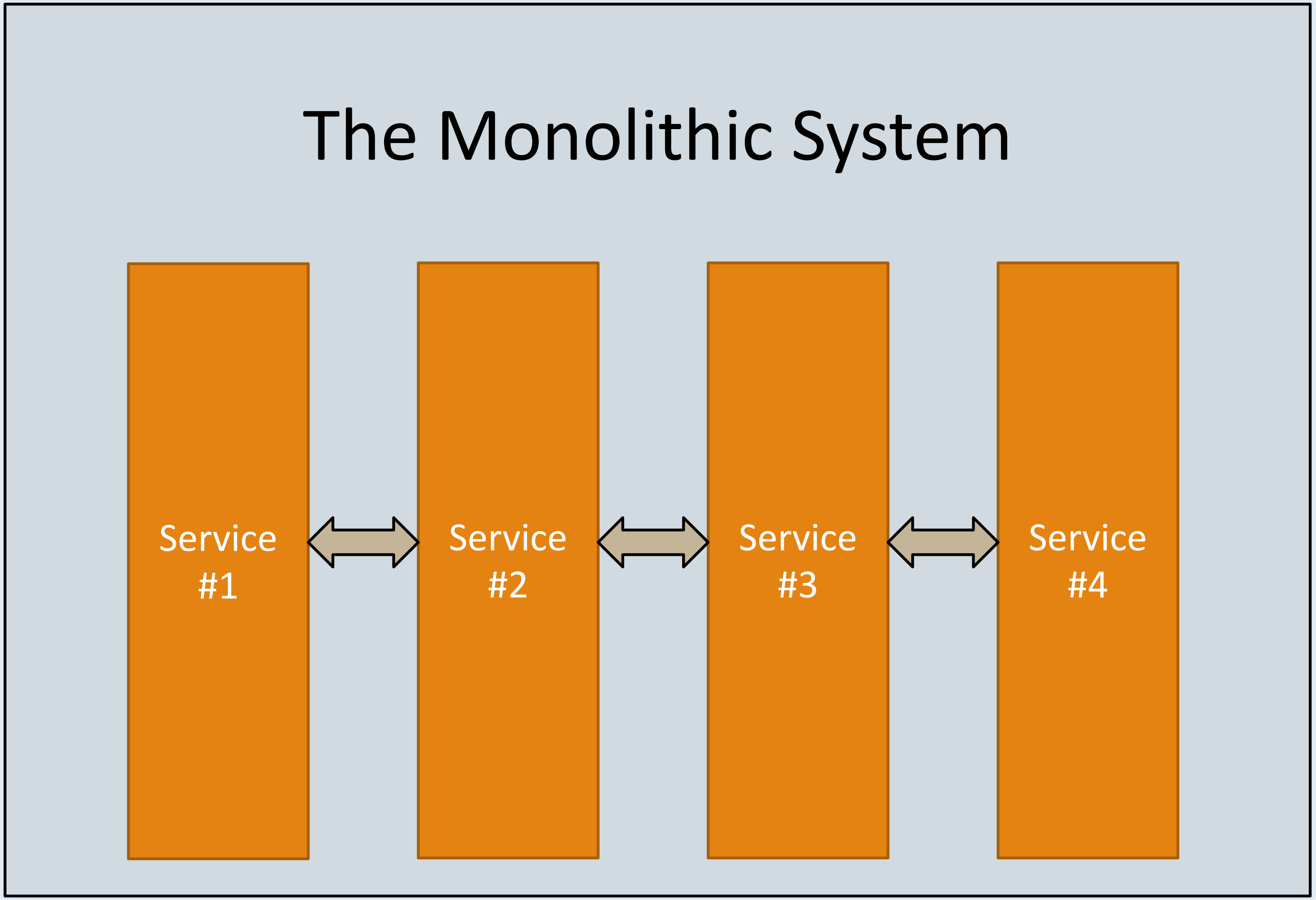
Right now we are cutting it *horizontally*

The Monolithic System

The Monolithic System

How about if we
cut it
Vertically ?

Services represent a complete, coherent task



How about if we cut it *Vertically*?

What are “Microservices” ??

Microservices are

Small, Autonomous services, that work together

- The *smaller systems* we talked about

Microservices are

Small, Autonomous services, that work together

- The *smaller systems* we talked about

How small is small?

- There's no clear definition
- *Lines of code* is not very apt to calculate
  - 100 lines of **C** code ≠ 100 lines of **Python** code
- Something that could be re-written in roughly two weeks time [2]
- Small enough, and no smaller [3]

Microservices are

Small, Autonomous services, that work together

- The *smaller systems* we talked about

Capable to exist autonomously

- Shall be independently deployable
- *Ideally* on a separate *host*
- All communications to outside world are via network calls
- Interface to communicate should *ideally* be technology agnostic

What do we gain by using Microservices ??

Technology Heterogeneity

Sometimes we choose technology that is the lowest common denominator

- Some things may be done better in some technologies
- A use-case may fit more to a NoSQL engine, than to an RDBMS engine

The learning curve for new technologies could be steep

- There may not be enough time for proper training of the staff
- It may be too risky to try them with little experience

The interfaces may restrain our choices

- What if a component we need to talk only knows RMI?
- We are forced to use Java elsewhere too !

Technology Heterogeneity

Since all the *smaller systems* are autonomous, they can be built using separate technologies

- As long as they conform to certain loose restrictions
- Like “being able to run on Linux”
- Lesser the restrictions, better it is

If we fail, we don’t lose much

- Since microservices are small in size, if we screw up, or realize that the technology change is adversarial, we still don’t lose much
- We can revert to old technology and rebuild the part from scratch

Low risk, more scope to experiment

- We can pick up technologies that we think are better, to build one or few services at a time
- If they work, we can do the same with others

Resilience

When something goes wrong in the monolith

- It can have a cascading effect
- One problem can bring down the whole system

Microservices are isolated from each other

- If one service is down, the others may still keep going
- Supports the idea of graceful degradation much better

Fault isolation and repair becomes easier

- The service that is down, needs to be looked into
- Smart logging can make it easier to find bugs, and repair them quickly

Ease of Deployment

In monolithic systems, changes are not easy

- Every change may involve a complete re-build, and then a re-deployment
- The application most probably will be offline for this time

So changes accumulate

- To avoid re-deployment, changes are accumulated till they become significant
- Higher the delta between releases, higher the risk

Since all microservices are deployed independently

- Changes to one can be deployed without affecting others (changes which are *not breaking* changes)
- If the new release doesn't work, a quick restore can be made to previous version

Time to market is reduced with microservices

- New changes can be rolled out easily, and with relatively lesser risk

Organizational Alignment

A small team is a happy team

- Smaller teams are generally more productive
- Reduces involvement of too many brains
- Better communication *within the team*

One microservice can be owned by one team

- No external involvement, teams can be responsible for the full lifecycle of the service, including operation and maintenance

Teams can be built around business contexts rather than technology

- One team can cater to one business subdomain, via one or more microservices
- Can also interact directly with the customers to get better feedback

Optimized for Replacability

Bigger systems are hard to modernize or replace

- Building a newer version from scratch may involve too much effort, and risk

Microservices are optimized for replacement

- Since they are small, it is easier to write them from the scratch and replace
- For example, a custom made single-sign on service can be replaced by sign-in via Facebook easily

What’s the catch then in using them ??

Not Magic Wands

So why don’t we all start using microservices?

- Just like any other Architectural Style, microservices have trade-offs
- Not everything associated with them make life easier

Not Magic Wands

So why don’t we all start using microservices?

- Just like any other Architectural Style, microservices have trade-offs
- Not everything associated with them make life easier

Network Calls are slower

- Out-of-process call, like HTTP request is always slower than in-process call like method invocation

Networks can fail

- It may be difficult to figure out if a service is down, or is out-of-reach

Networks are insecure

- If the data travels over a network, say internet, it needs to be properly guarded against adversaries

Not Magic Wands

So why don’t we all start using microservices?

- Just like any other Architectural Style, microservices have trade-offs
- Not everything associated with them make life easier

More *hosts* to manage

- Typically each microservice runs on a different host
- A host could either be a physical machine, a virtual machine or even a docker container
- More microservices mean more hosts to manage

Not Magic Wands

So why don’t we all start using microservices?

- Just like any other Architectural Style, microservices have trade-offs
- Not everything associated with them make life easier

Distributed Transactions

- With one database or process, transactions are easier to implement
- A rollback in case of failure is easy to achieve
- With multiple processes and database instances, each keeping track of some part of an overall business transaction, it is difficult to revert changes in case of failures

Not Magic Wands

So why don’t we all start using microservices?

- Just like any other Architectural Style, microservices have trade-offs
- Not everything associated with them make life easier

Distributed Monitoring and Logging

- Logging and monitoring one host is significantly simpler than a cluster of hosts
- Without automation, things can quickly become unmanageable

Modelling the system – business perspective

So we’ve decided to use the microservices style

So we’ve decided to use the microservices style

Where to start then?

So we’ve decided to use the microservices style

Where to start then?

Remember we talked about *cutting vertically* ?

So we’ve decided to use the microservices style

Where to start then?

Remember we talked about *cutting vertically* ?

Ideally, one business task is represented by one service

So we’ve decided to use the microservices style

Where to start then?

Remember we talked about *cutting vertically* ?

Ideally, one business task is represented by one service

So understanding the domain may be a good starting point

So we’ve decided to use the microservices style

Where to start then?

Remember we talked about *cutting vertically* ?

Ideally, one business task is represented by one service

So understanding the domain may be a good starting point

How to model these business aspects?

Bounded Contexts

What does a business comprise of ?

- Departments or sections
- For example, the Finance Department, the Store, the H.R. Department, the I.T. Department etc.
- A particular department, say Finance, may provide certain information to other departments, say I.T., via *interfaces*
- The department may share some information with others, while at the same time, keep some information private with itself
- For example, the Finance Department may have an interface, by which I.T. Department can pull up *selected finance* data about an employee, and show it on an internal portal

These departments or sub-domains are known as *Bounded Contexts*

- A Bounded Context represents a specific set of responsibilities, guarded by certain boundaries
- Bounded Contexts themselves can be made up of smaller, sub-contexts

Bounded Contexts

Bounded Contexts have *models*

- Every Bounded Context have models associated with them
- For example, the H.R. Department may have a model that keeps information of an employee, storing details such as number of dependents, current grade, and leave balances
- The Finance Department will also have a model of an employee, storing details such as salary, dearness allowance, savings declarations etc.
- The two departments may share a common aspect of the model, say Employee Id

Bounded Contexts share information via interfaces

- Imagine that the Finance Department needs to know about the unpaid leaves an employee took in the last month
- The H.R. Department may expose an interface, which, on presenting with an Employee Id, may give out unpaid leave details
- Note that the H.R. Department may decide what aspects of the employee model to keep hidden, and what to expose

Deciding Service Boundaries

The most tricky part in designing a microservice oriented system, is probably the choice of service boundaries

- Often we are tempted to choose boundaries that are *technical* and not *business oriented*
- Bounded Contexts can help
- A Bounded Context could be a good choice to draw service boundary

Don’t finalize the boundaries too soon

- It is extremely difficult to get the boundaries right in the first attempt
- One can make mistakes in doing so
- If you are unsure if a split makes sense or not, *stay on the monolith side* for some time – i.e. merge these contexts (or don’t split them at first place)
- The more time one spends with the businesses, the more could be garnered about the Bounded Contexts

Handling dependencies

Irrespective of how much we try to segregate our services, *no service can exist* in isolation

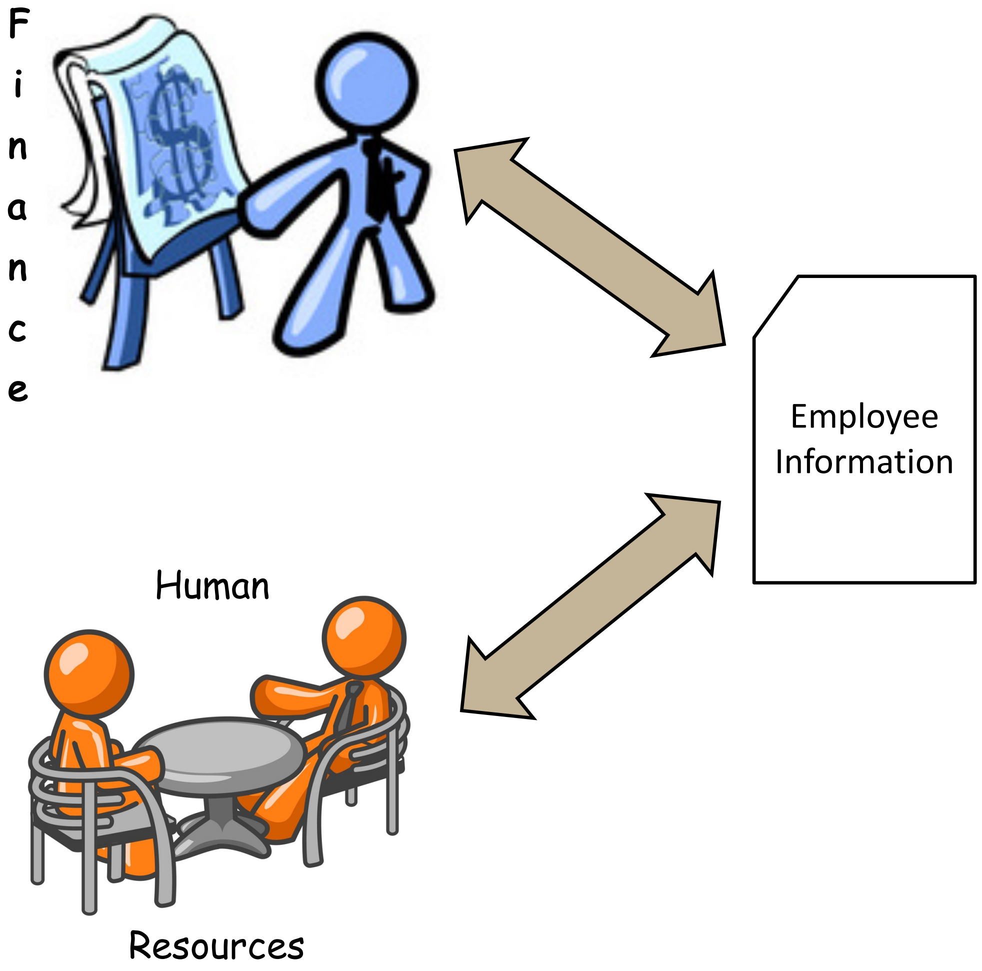
- Taking the example of Finance and H.R. Department, we may be tempted to keep one *Employee Table* in the Database, keeping all the required information for both departments
- But the Employee Table is now a dependency for both Departments
- If a field needs to be added to the Employee table in future, who decides? In other words, *who owns the Employee Table*? The H.R. Department or the Finance Department?

Database oriented dependencies are most common in monoliths

- Not a trivial job to break these dependencies
- May depend on case to case basis, what's the best we can do?

Breaking Data Dependencies – An example

Breaking Data Dependencies – An example

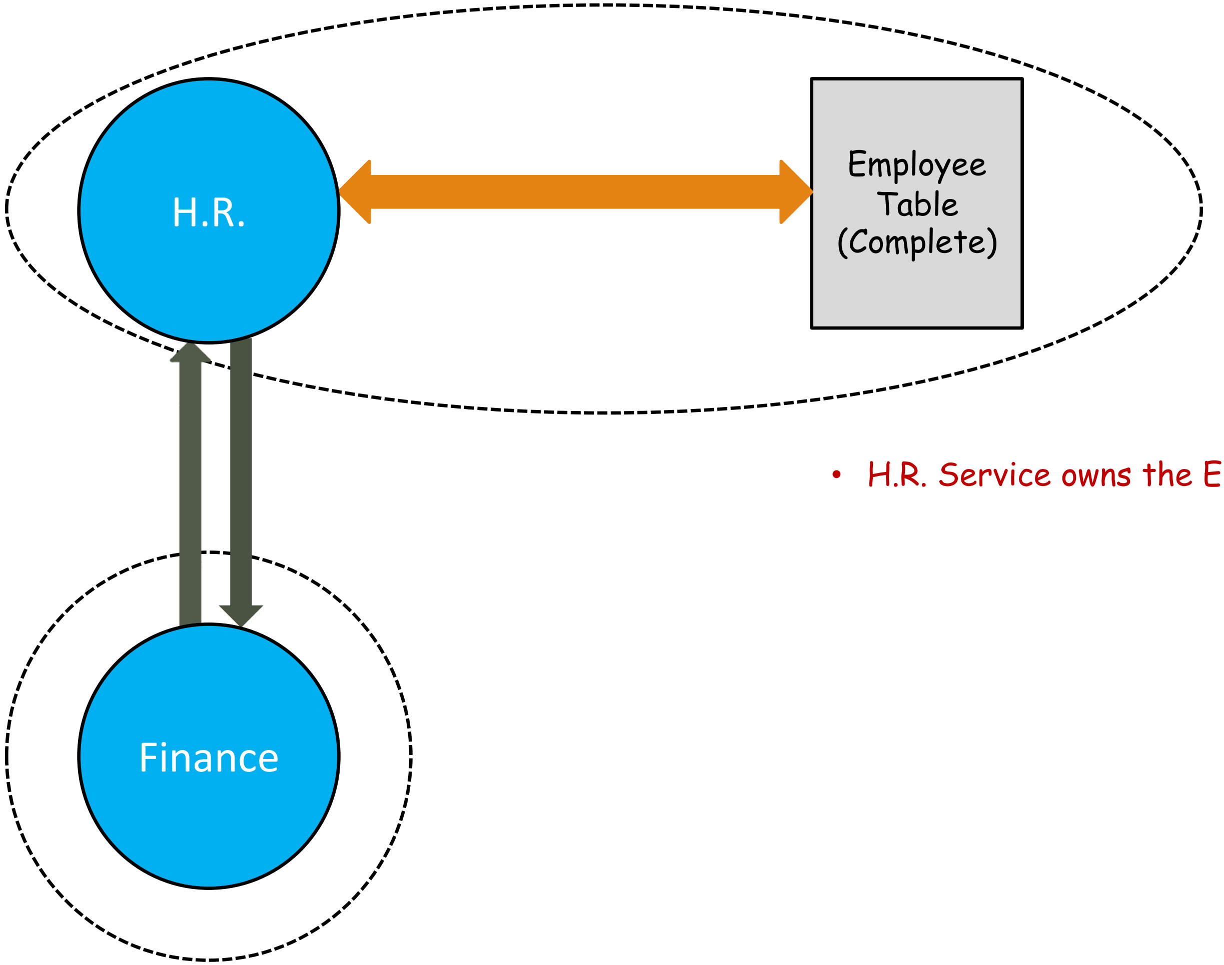


Finance

Employee Information

Human Resources

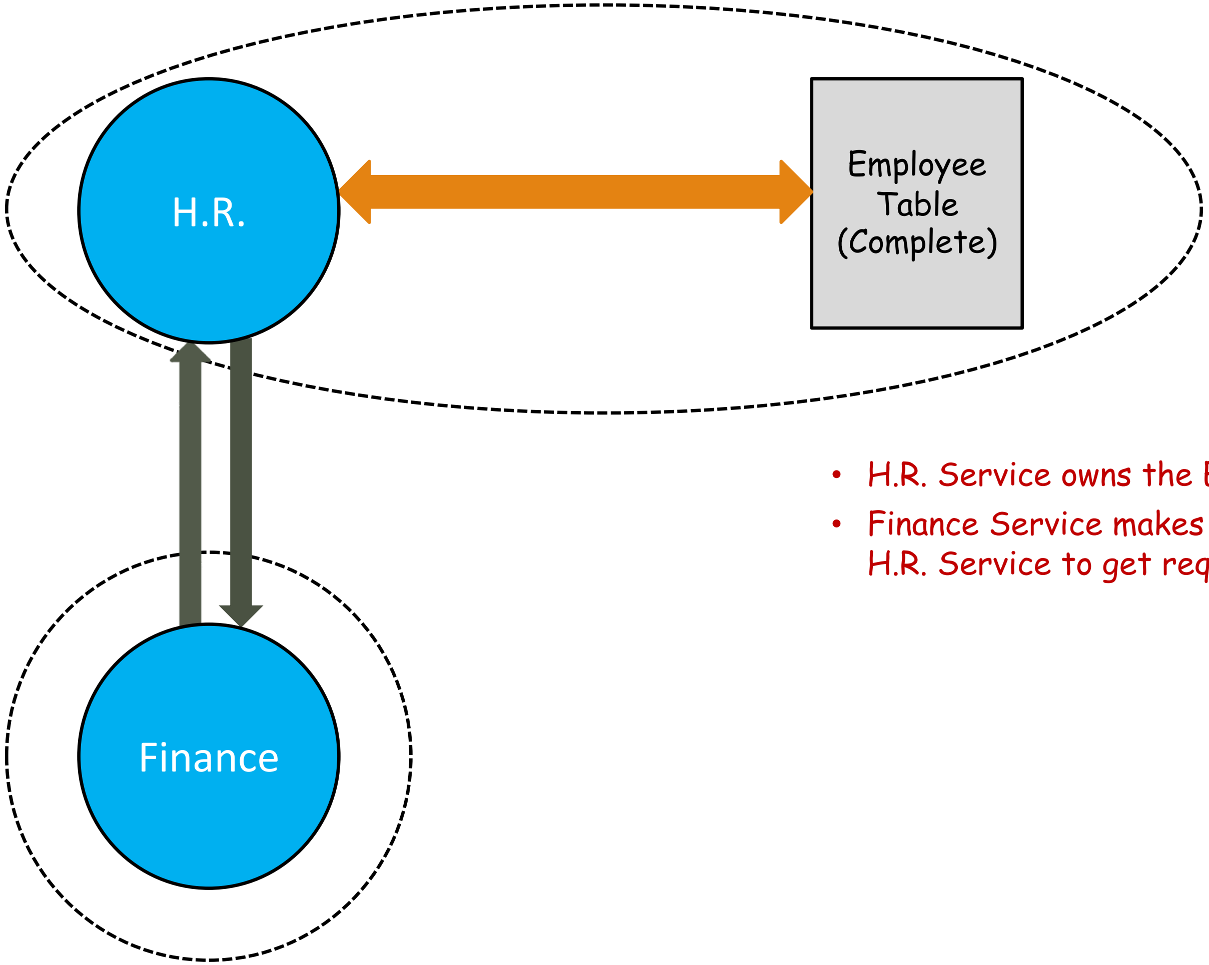
One possible way to draw service boundaries



- H.R. Service owns the Employee Table

---------- Service Boundary

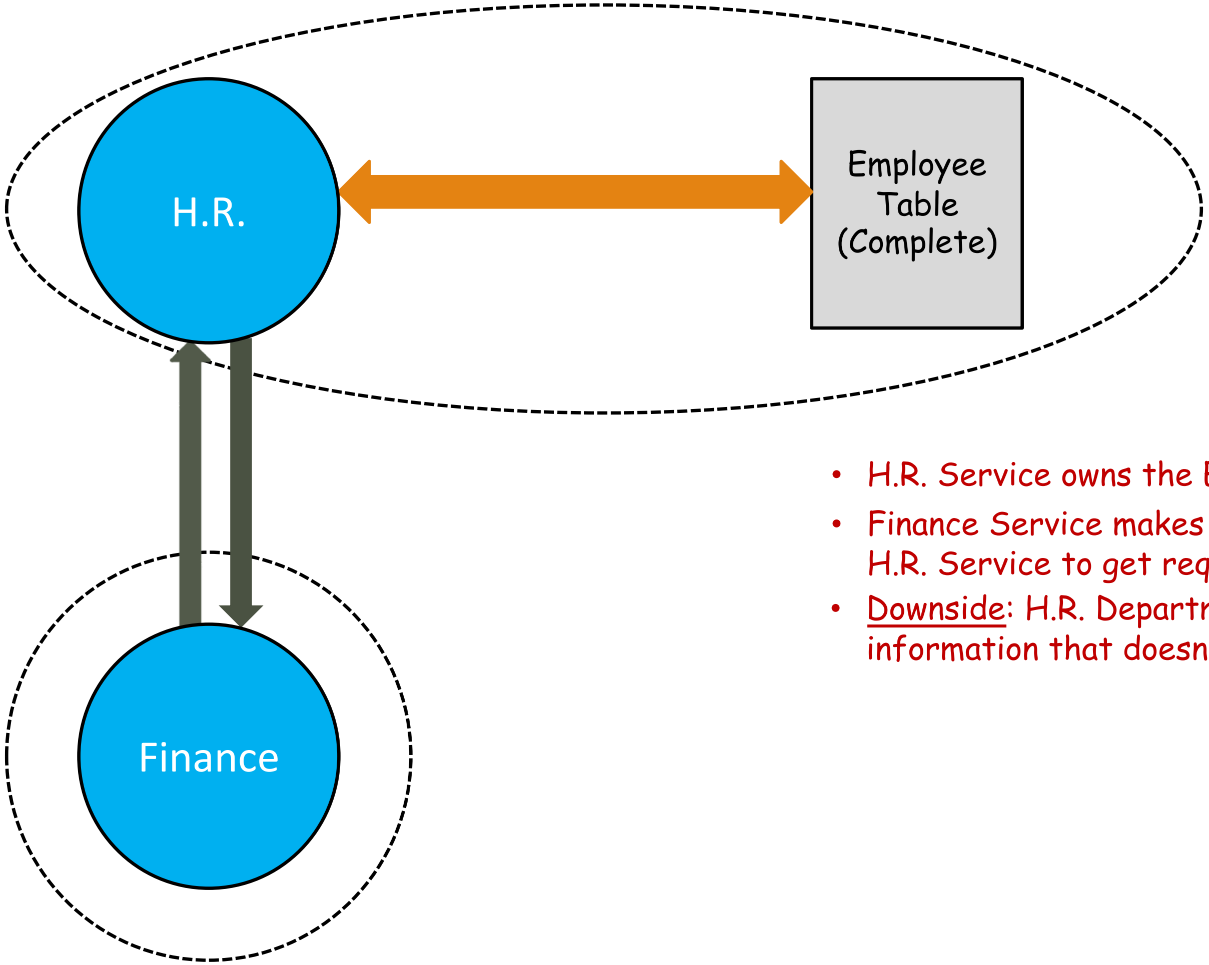
One possible way to draw service boundaries



- H.R. Service owns the Employee Table
- Finance Service makes an API call to the H.R. Service to get required Information

---------- Service Boundary

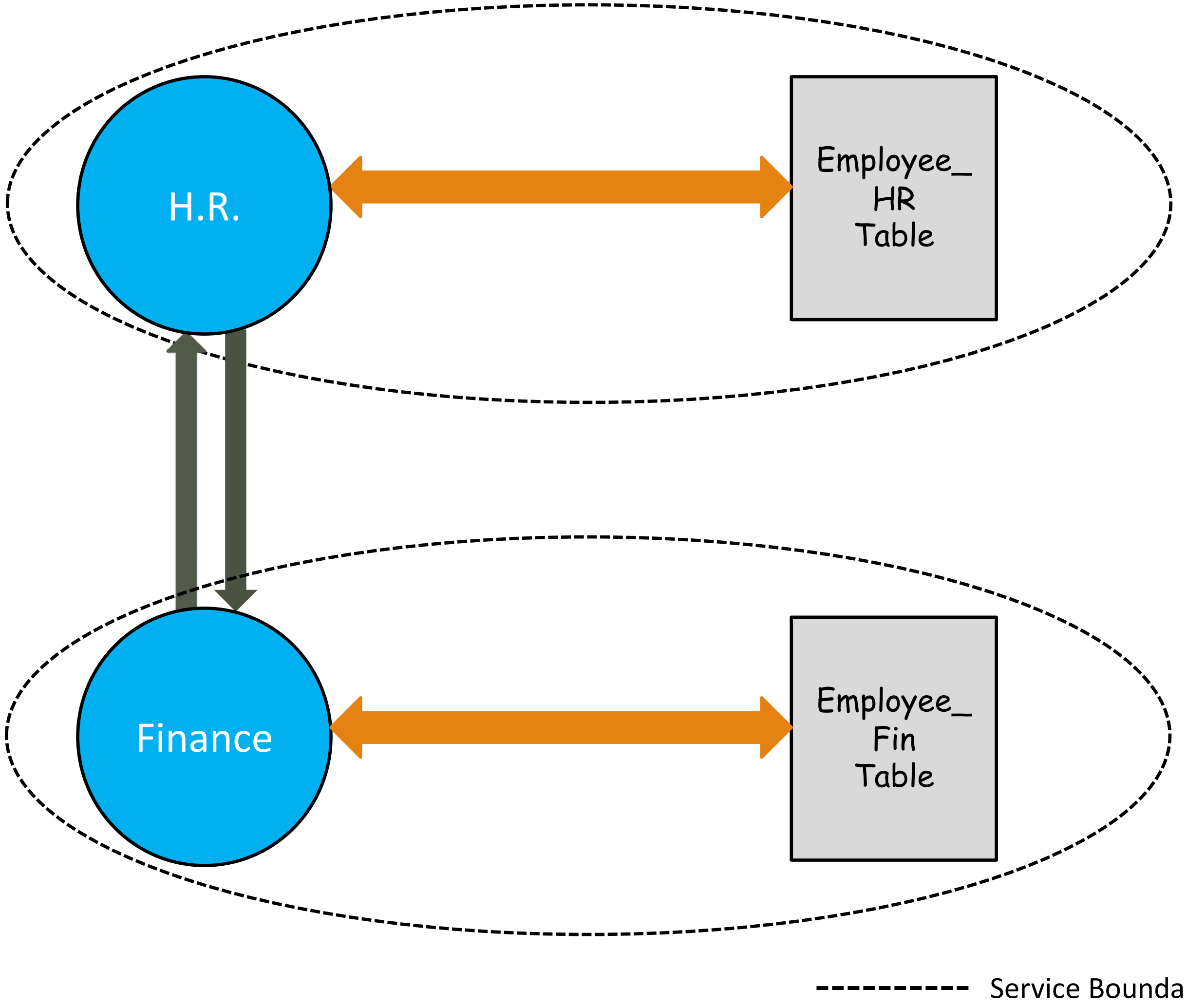
One possible way to draw service boundaries



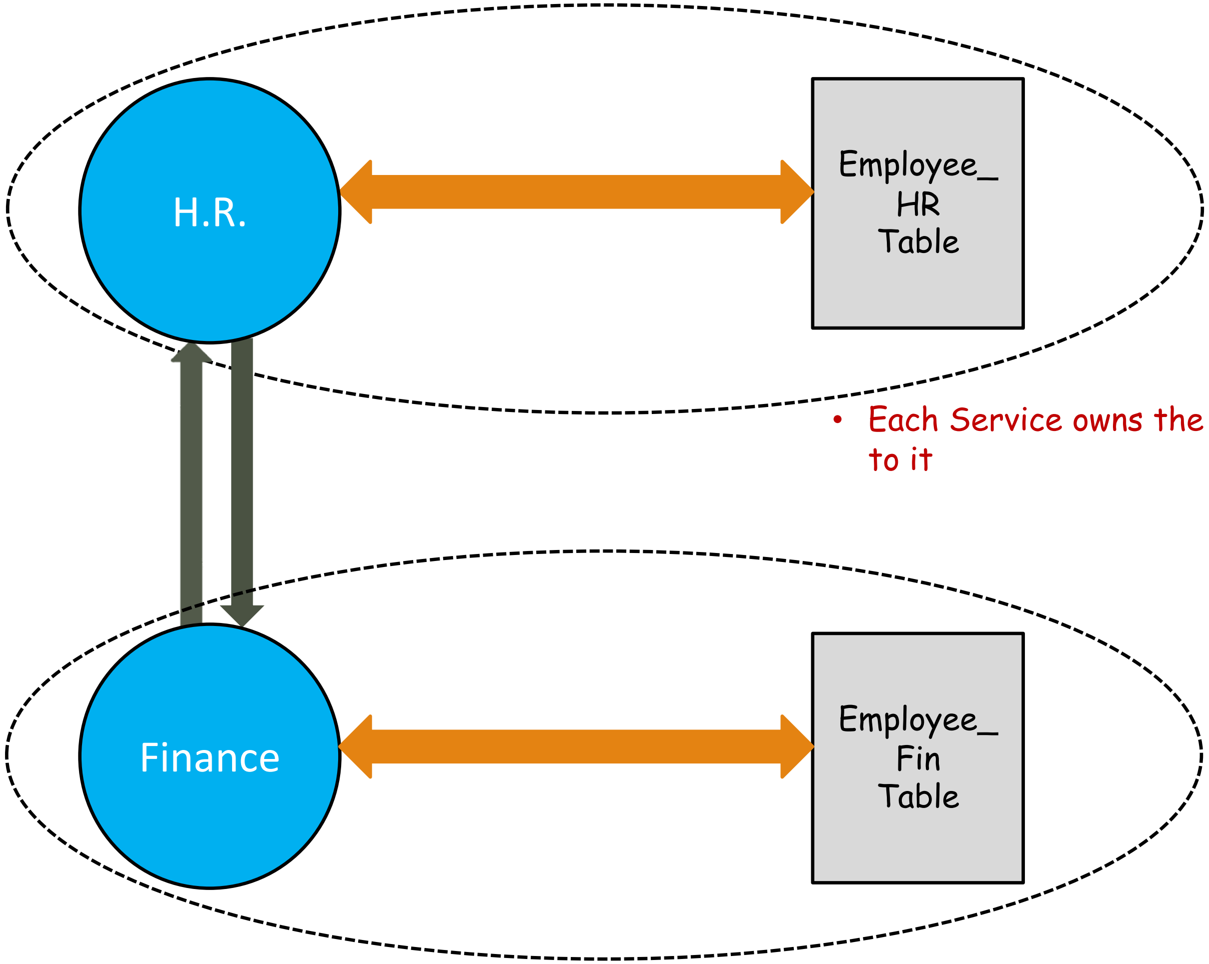
- H.R. Service owns the Employee Table
- Finance Service makes an API call to the H.R. Service to get required Information
- Downside: H.R. Department is exposing information that doesn't belong to it

---------- Service Boundary

Another possible way to draw service boundaries



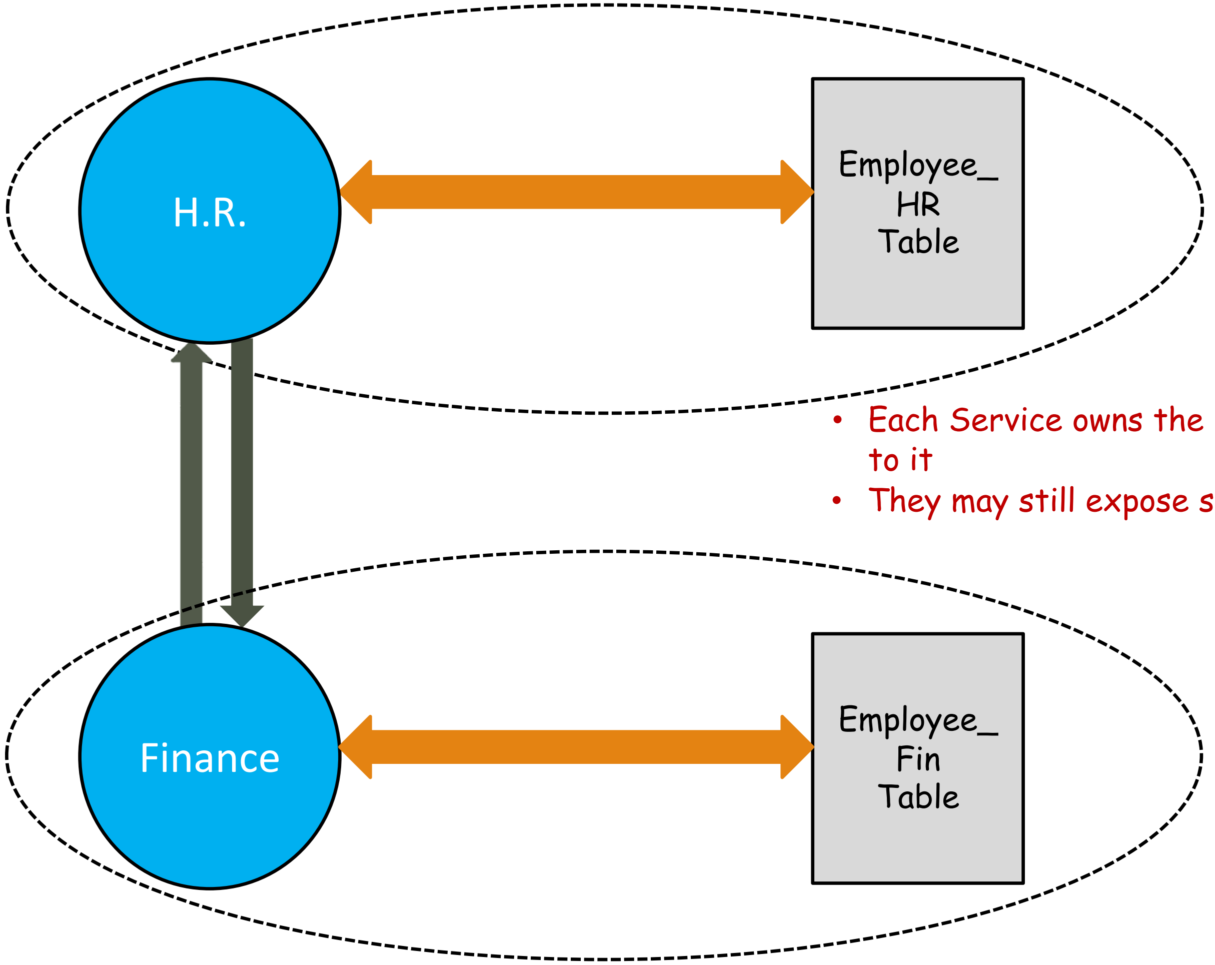
Another possible way to draw service boundaries



- Each Service owns the data that belongs to it

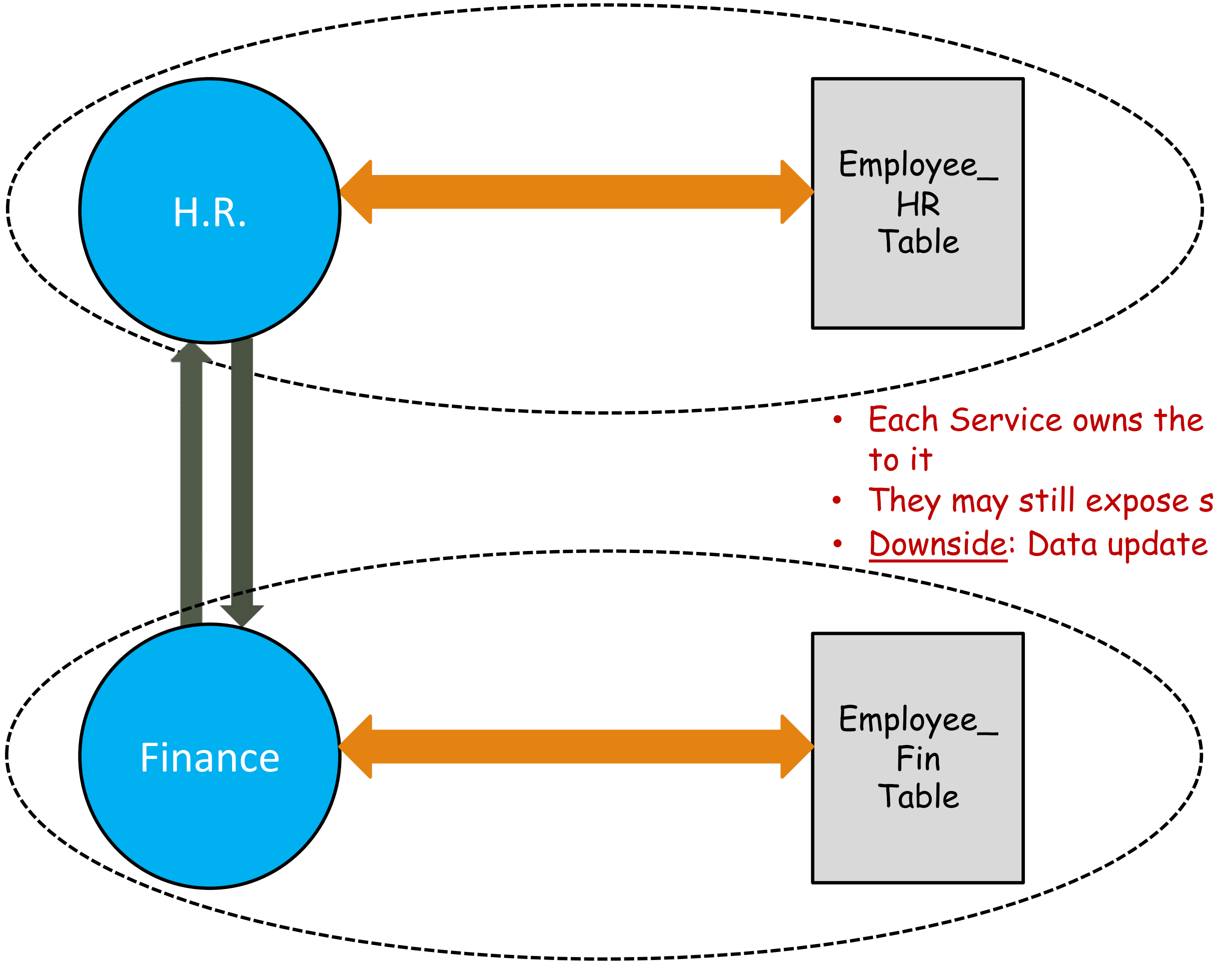
---------- Service Boundary

Another possible way to draw service boundaries



- Each Service owns the data that belongs to it
- They may still expose some data

Another possible way to draw service boundaries



- Each Service owns the data that belongs to it
- They may still expose some data
- Downside: Data update and sync issues

---------- Service Boundary

Deploying and Testing Microservices

Let’s say we finalized one or more service boundaries

Let’s say we finalized one or more service boundaries

And we found/formed teams willing to build them

Let’s say we finalized one or more service boundaries

And we found/formed teams willing to build them

Are we done then?

Let’s say we finalized one or more service boundaries

And we found/formed teams willing to build them

Are we done then?

How will these services be deployed?

Let’s say we finalized one or more service boundaries

And we found/formed teams willing to build them

Are we done then?

How will these services be deployed?

How will they talk to each other?

Let’s say we finalized one or more service boundaries

And we found/formed teams willing to build them

Are we done then?

How will these services be deployed?

How will they talk to each other?

How can we test them? In isolation and in action?

Continuous Integration and Delivery

Goal:

- Integrate different seams into one, and create an overall build of the product
- A concept independent of microservices, yet can be very helpful if adopted with them

Artifacts:

- One or more artifacts are produced at the end of the *Build Pipeline* [6]
- A Test Suite is created, which is run in an automated fashion on the artifact(s)

Red builds need to be fixed

- A *Red Build* is a build that failed to produce the artifact(s) it was supposed to
- In Continuous Integration / Continuous Delivery, a failed build is resolved at high priority
- Contrast that to a series of commits, coagulating in a repository, which is then built once a month

Jenkins [7] and Bamboo [8] are two well known CI Tools

Artifacts

Artifacts produced by Continuous Delivery may be technology specific

- Java has *JAR* and *WAR* files
- Ruby has *Gems*
- Python has *Eggs*
- The artifacts themselves may not be deployable directly
  - We may have to use some level of automation, say Puppet Scripts, to put them in place
- The drawback here is if we allow such artifacts, we may end up having to manage multiple platforms to support each kind of artifact

Operating System level artifacts

- One way to avoid this, is by using Operating System level artifacts
- For example, we may create an RPM file for RedHat or CentOS based systems, or a DEB package for Ubuntu, or an MSI for Windows
- Downside: it is not that simple to find dependencies and create these artifact for multiple platforms

VM Images as artifacts

One way to deploy services, is via VM images

- Create a custom deployment image which can be spawned
- Typically, choose an operating system, say Ubuntu, install the tools required for running the service, say Apache and MySQL, and then save this base image in an Image Repository
- At the end of the Delivery Pipeline, an image is spawned, and the service is deployed on it
- Abstracts all the technologies, even the operating system in use
- A Virtual Machine player, like Virtualbox, can *play* the image, and we are done !

Downside: No standardization !

- An image made to run on AWS, won’t run on VMWare
- Building images for multiple platforms, in an automated fashion, is still not easy

Packer [9] is a tool which can make our lives easier

- Supports building images for multiple platforms, from single source configuration

Testing the services

Unit Tests

- Test functionalities local to a sub-system or microservice
- For instance, calls to certain methods, with expected returns lie in these categories
- Quite fast in general, several tests per minute can be run

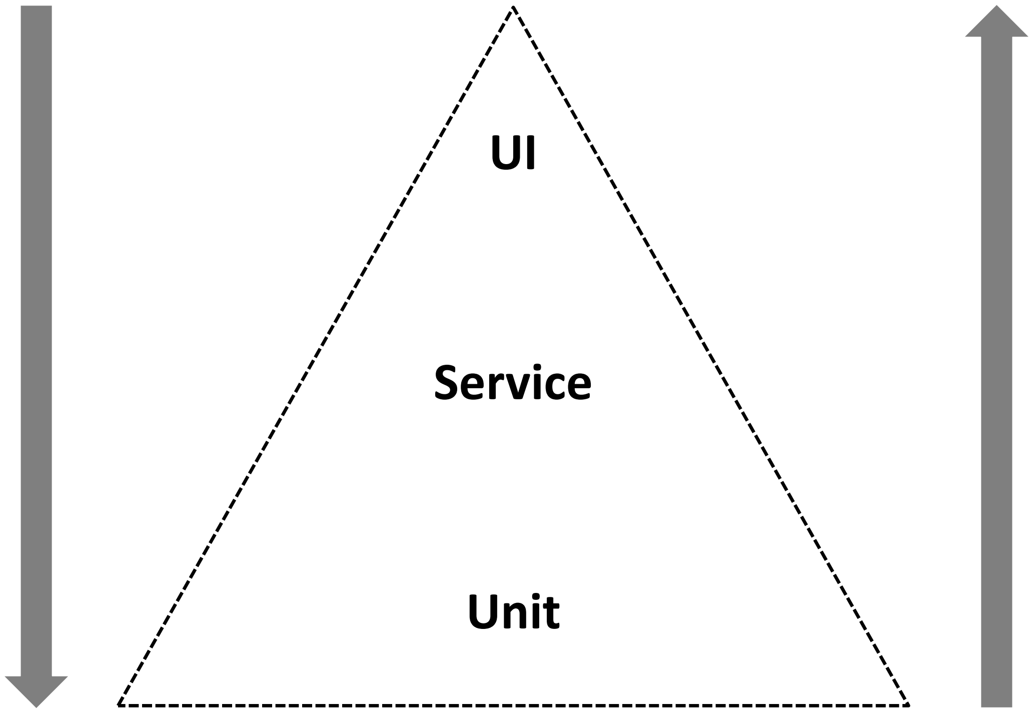
Service Tests

- Test functionalities expected to be fulfilled by a service
- May include tests to validate interfaces exposed by a service, e.g., the output format
- Fast enough generally, unless the Service needs to make downstream calls
- We may need to *mock* behaviours for any downstream processes

UI Tests or End-to-End Tests

- Cover cross-service aspects – generally one full use-case

The Test Pyramid



UI

Service

Unit

Mike Cohn.'s Test Pyramid [5]

Faster
Better isolation

Increasing scope
More confidence

- As we move down, tests cover only specific aspects of the system
- However, they are faster, and can help pin-point the problem
- If a test up in the pyramid passes, it gives more confidence about the overall well-being of the system

Consumer driven tests

Services are consumers as well as producers

- A service, in order to perform its tasks, may invoke other services
- Remember the H.R. and Finance example
- If we choose to use the first model, then the Finance service becomes a *consumer* of H.R. service

Consumer-driven Contracts (CDCs)

- CDCs are formal expectations of a consumer, from a service [3]
- If captured fully, they can be baked into the CI build process of the producer
- If a release violates the contract, i.e., produce *breaking changes*, it never gets deployed

Tooling to help CDT

- **Pact** [11] is a tool developed originally at RealEstate.com.au which now supports Ruby, JVM and .Net for Customer driven testing

Blue/Green Deployment

The Blue/Green Deployment model is used in case we have little confidence over the new release

- After deploying a new instance of a service, do not direct production load on it straightaway
- Run a smoke test suite on the newer service instance(s)
- Detect any issues with deployment environment
- After the tests succeed, allow production traffic on the new instance
- Do keep the older ones alive for a while for any plausible rollbacks

Be prepared for rollbacks

- If something goes wrong during the testing, or if the service doesn't perform as per expectations, the older version is rolled back

Canary Releasing – a slight variant

- Keep both versions in production, for a longer time, and evaluate them side-by-side; complete roll-out happens when there is enough confidence

“All successful applications have an architecture that sucks.”

Stefan Tilkov ‘2014

Connectors for Components

Architecture involves designing proper components, and proper *connectors* to connect them

- Connectors are as much bit important as the components
- Even the most sophisticated components may look like clowns, if they are connected with horrible connectors

Microservices bring with them, a problem

- Since services *hear* and *respond* over networks only, we are essentially choosing an unreliable connector to connect our components
- Networks can fail, and will fail at some point of time – there is no way we can hide from this reality

So how do we integrate our services in this world?

- Build applications that are resilient to failures
- Expect failures, use failures as a part of the overall testing process

Synchronous vs Asynchronous Calls

Synchronous Calls

- *Service #1* calls *Service #2* and blocks till it receives a response
- Simple to implement
- Results are known straightaway

Downsides of Synchronous Communication

- What if *Service #2* is down?
- There can be a Network outage instead

Synchronous vs Asynchronous Calls

Asynchronous Calls

- *Service #1* requests *Service #2* for a response
- *Service #1* doesn't wait for the response, but keep performing a periodic check to see if the response has arrived
- When it does, *Service #1* processes the same
- More robust than waiting for *Service #2* to respond, generally a better choice for connecting services
- Good for long-wait calls

Downsides of Asynchronous Communication

- Need of extra resources, like a Message Queue
- Slightly more complicated to implement
- How do we know if we'll get a response at all? What if there is a failure?

Orchestration vs Choreography

Orchestration

- There is one master service, which calls other services to achieve the overall goal
- The other services do their job as per instructions from the master
- Much like how a conductor gives directions in the form of signs, and musicians play accordingly

Orchestration vs Choreography

Choreography

- Each Service is aware of what is has to do
- An event, created by one service, can be acted upon by one or more services, in their own way
- Easier to decouple services, as the overall process can be asynchronous too
- Like a dance performance, pre-choreographed, where everyone knows what to do and when

Example – Inter-service communication

Example – Inter-service communication

- Let’s say the organization decides to give all its employees above 50 year of age a monetary bonus for New Year. The organization also decides to send a box of sweets to all these employees.

Example – Inter-service communication

- Let's say the organization decides to give all its employees above 50 year of age a monetary bonus for New Year. The organization also decides to send a box of sweets to all these employees.
- Let's say the H.R. Department initiates the process, by finding out the employee ids of all such employees.

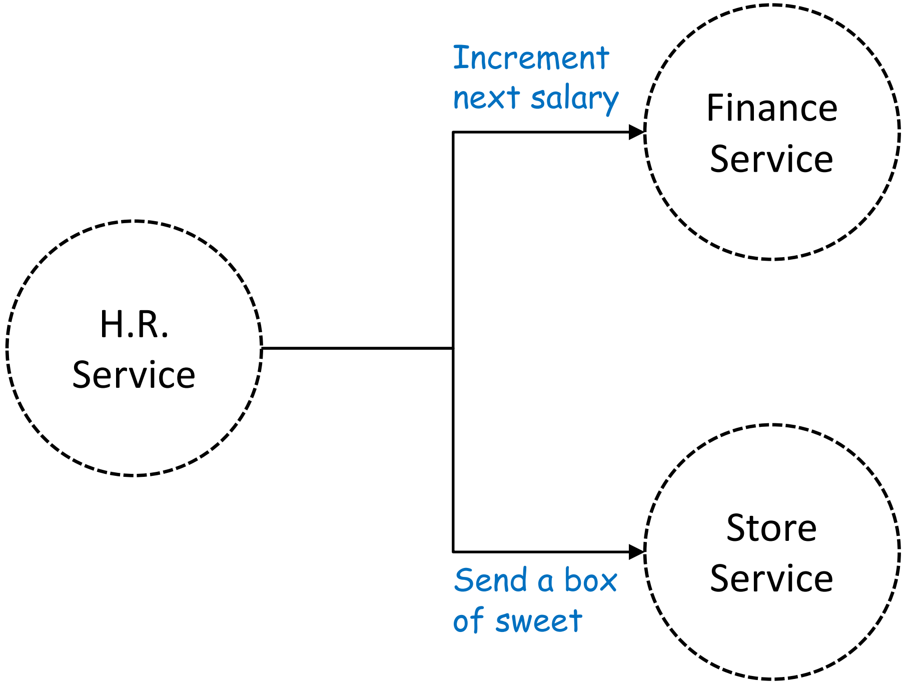
Example – Inter-service communication

- Let's say the organization decides to give all its employees above 50 year of age a monetary bonus for New Year. The organization also decides to send a box of sweets to all these employees.
- Let's say the H.R. Department initiates the process, by finding out the employee ids of all such employees.
- Now the Finance Department needs to make an increment in their next salary.

Example – Inter-service communication

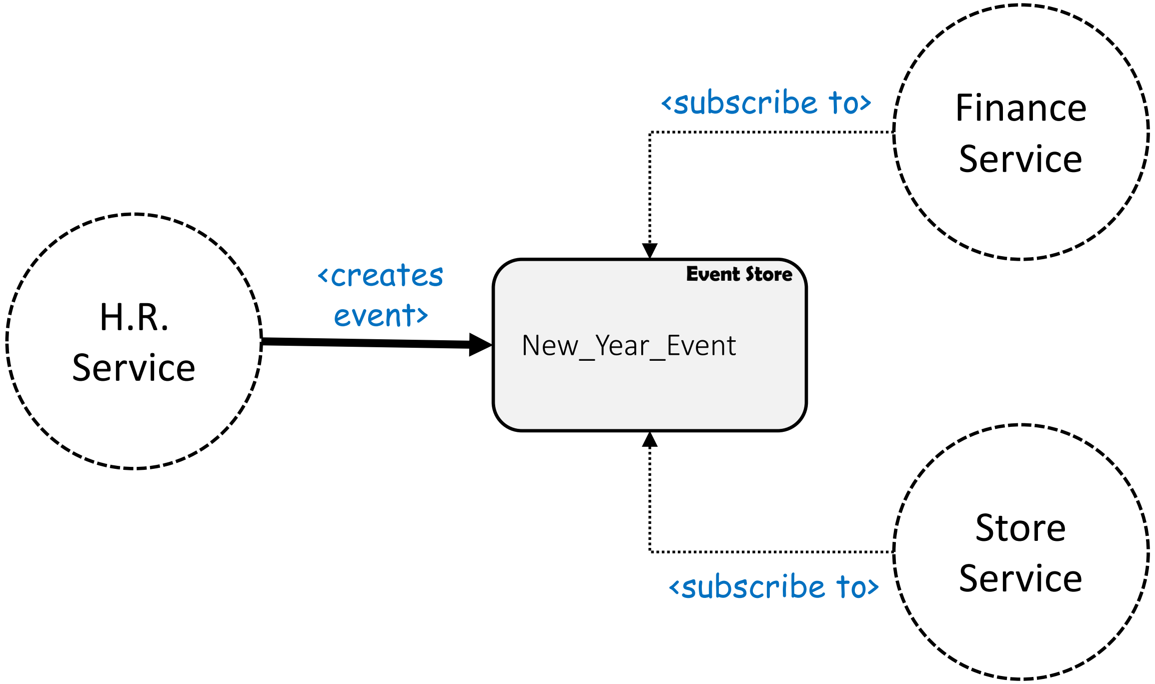
- Let's say the organization decides to give all its employees above 50 year of age a monetary bonus for New Year. The organization also decides to send a box of sweets to all these employees.
- Let's say the H.R. Department initiates the process, by finding out the employee ids of all such employees.
- Now the Finance Department needs to make an increment in their next salary.
- The Store Department needs to hand them over a box of sweet from the store (let's assume the organization is *sweet* enough to stock sweets too !!).

Using Orchestration



The H.R. Service is the Orchestrator, which tells other services to act on an event

Using Choreography



The H.R. Service creates an event with a JSON list of Employee Ids, and other services act upon it, as per their convenience

Monitoring the services

So we’ve deployed the services we built

So we’ve deployed the services we built

… but as we talked about, getting it right is not easy

So we’ve deployed the services we built

… but as we talked about, getting it right is not easy

… specially in the initial phases

So we’ve deployed the services we built

… but as we talked about, getting it right is not easy

… specially in the initial phases

So we need to keep an eye for any problems

So we’ve deployed the services we built

… but as we talked about, getting it right is not easy

… specially in the initial phases

So we need to keep an eye for any problems

Monitoring microservices may be more complex than a monolith

So we’ve deployed the services we built

… but as we talked about, getting it right is not easy

… specially in the initial phases

So we need to keep an eye for any problems

Monitoring microservices may be more complex than a monolith

Logging plays an important role here

One vs *Many* to monitor

Monitoring the *monolith*

- With monolith applications, monitoring is somewhat simpler – monitor the monolith
- This may mean running tools to collect machine level statistics as well as collecting information about the application operation itself
- If something goes wrong, we have to mine the monolith to find the problem

Monitoring the mess

- Microservices bring with them, another issue – how to monitor the different *sub-systems* effectively?
- We need tools that can collect statistics on multiple hosts, as well as statistics that are application level, i.e. cross-cut more than one services
- If something goes wrong, the problem may span across the boundaries of services, or can either be a tethering issue, due to a problem in the deployment environment

So how to do we deal with this?

- Common Sense: build the big picture from the smaller pieces available

Monitoring across different levels

We need statistics at varying levels to be able to diagnose and solve problems

- We need statistics about the host on which service is running
- We need statistics about the platform, e.g. the Web Server, on which the service runs
- We need statistics about the application as a whole or a part of it

Machine level statistics

- We need to monitor the host on which services are deployed
- As a minimum, at least a periodic recording of the CPU Usage and Memory Usage is essential to making inferences about the service
- A high CPU usage on one or more hosts, and not on others, deploying the same service(s), may indicate a problem at Load Balancing stage

Monitoring across different levels

We need statistics at varying levels to be able to diagnose and solve problems

- We need statistics about the host on which service is running
- We need statistics about the platform, e.g. the Web Server, on which the service runs
- We need statistics about the application as a whole or a part of it

Platform level statistics

- We can’t produce everything we use, in-house !
- We’ll be using a specific mix of technologies over which our services run
- Probably the most common example is a Web Server like Apache or Tomcat
- We need to monitor that too
- Most of the servers are capable of producing detailed log of events that occur, such as error logs, access logs etc.
- They can quickly fill disk space though – log rotation (keep logs of up to *n days*) is a practical option

Monitoring across different levels

We need statistics at varying levels to be able to diagnose and solve problems

- We need statistics about the host on which service is running
- We need statistics about the platform, e.g. the Web Server, on which the service runs
- We need statistics about the application as a whole or a part of it

Application level statistics

- The application itself may have a number of things to monitor
- If nothing else, monitoring the Response Times for the requests seems a bare minimum for any health checks or debugging
- The services may be written to write their own log entries, or send periodic information to a logger service, that keeps track of application level statistics
- These stats may be the most handy, when we discover a bug in one or more services

Saviours of the mess

Lots of scattered data

- Even if we optimize our monitoring process to collect just the right amount of statistics, we may still be looking at lots of data
- To complicate the problems, the data could be distributed among different hosts
- Aggregating this data, and then abstracting it for sensible viewing may require fetching log files from multiple sources and consolidating them

Thankfully, there are tools for help

- **Nagios** [14] is a state-of-the-art tool to collect statistics on single or multiple hosts
- There are *ssh multiplexers* which can connect to multiple machines in an automated fashion, and run commands which can aggregate logs and then pull them to a central location for consolidation
- **ClusterSSH** [12] and **pdsh** [13] are two common tools which can do that for you

Saviours of the mess

Organizing the haphazard

- Ssh multiplexers can fetch data from different places to make this job easier, but you still need to consolidate the data to be presented in a concise fashion
- The logs that are retrieved, may follow different log patterns

Tools to parse and present logs

- **Logstash** [15] can parse multiple log formats, can apply transformations on them, and then send them downstream for further investigation
- Another popular tool which can allow searching through logs, restrict time frames, and even generate helpful graphs is **Kibana** [16]
- **Graphite** [17] is a tool which exposes a very simple API to collect and aggregate metrics, or drill down wherever required

Synthetic Monitoring

How often shall we check up on our systems?

- We may not know what a healthy system looks like, unless we actually put a load on it, see its performance in real-time
- Finding out exact values, or ranges, to comment on a service's health may require putting production load or *synthetic loads* on it, and analyze the results

Monitoring systems via synthetic loads

- Instead of monitoring the system regularly to see if "everything is good", we can periodically run *synthetic business transactions* or put synthetic load on them, and scan the results for any anomalies
- Synthetic monitoring is not an alternative to collecting regular stats though
- We may not be able to track a problem, if detected during the synthetic transaction, if we are not collecting the right information beforehand

Following the trail

How can we track problems spanning across multiple services?

- Use a unique ID, generated at the time a request enters the system
- Keep passing this id to all stages (service invocations) through which a request goes through
- These are called **Correlation IDs**

Using Correlation IDs

- Every service that receives a call from a consumer, receives the Correlation ID as well
- They can now be put in all log entries, made for serving this request
- On an error, this ID can be traced through the log, to separate event track for the particular request

Tracing the calls

- **Zipkin** [18] is a tool which can trace calls across multiple hosts, and can present a UI to inspect inter-services calls

Securing our microservices

Security is an important issue for all applications

Security is an important issue for all applications

… even more so, if they are distributed in nature

Security is an important issue for all applications

… even more so, if they are distributed in nature

Can a service trust the invoker’s credentials?

Security is an important issue for all applications

… even more so, if they are distributed in nature

Can a service trust the invoker’s credentials?

Is the communication safe from masquerading or tampering?

Authentication & Authorization

Authentication

- Authentication is the process by which we confirm that an entity is indeed the one it claims to be
- How can one prove his/her identity?
- The most common way to do so is via a *username-password* pair

Authorization

- Authorization is the process, in which an entity is granted certain privileges to access certain resources or facilities
- Generally, authorization is preceded by authentication
- Typical use case involves showing a user information related to his/her account (giving access to certain, out of all information available)

Need for authentication/authorization

How does a service know who’s the invoker?

- A service should be able to establish the identity of the consumer
- A consumer could be *an external user*, say a human communicating via a web browser, or *another service* from within or outside the system boundaries

Authorizing access to data

- The most common reason for a consumer to invoke a service, is to view or modify some data, owned by the service (or by downstream services)
- Without proper authorization, a service can not allow read and/or write access to the data

But signing in at multiple places is a nuisance

- Since all our services are *supposedly independent* and *loosely coupled*, this may mean each service wanting the consumer to authenticate itself before giving it any access
- Just imagine being asked to login every time you click on a link on a web page !

Single Sign-On (SSO)

Using the same *sign-in* for all

- Multiple services may share the same authentication requirements
- Instead of asking the user to login separately, an agent can perform the sign-in procedure once, and the services can then seek authorization information from the agent for all future calls

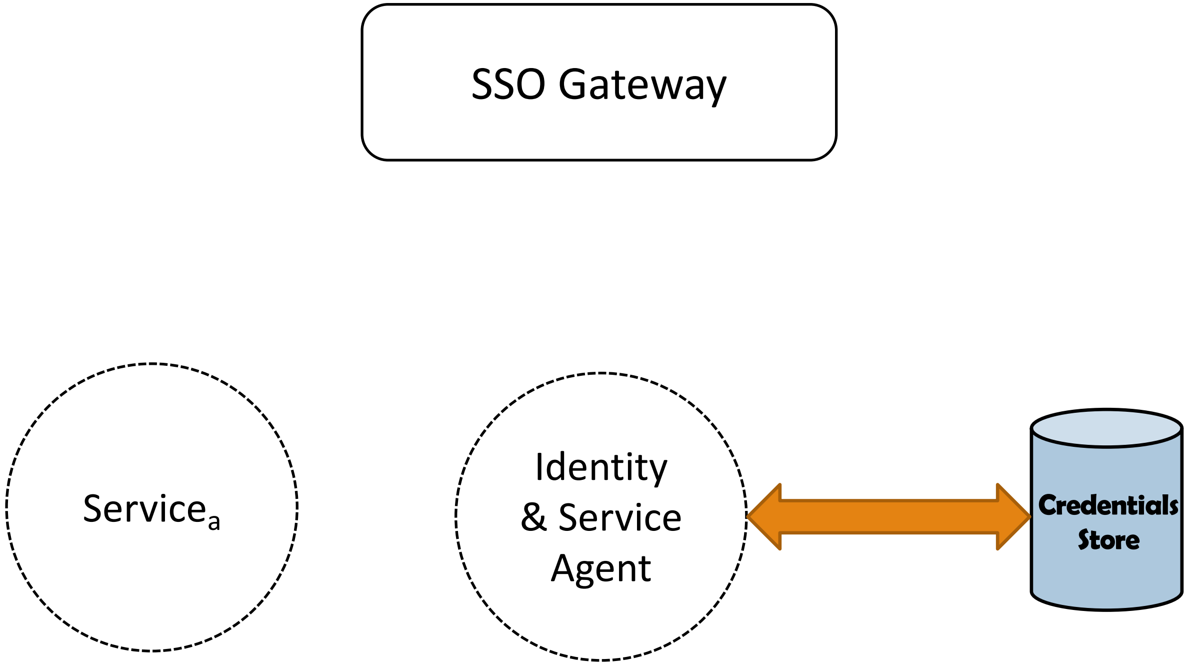
Identity Provider and Service Provider

- Typically, the Identity Provider is responsible for performing the authentication
- Once the authentication succeeds, the Service Provider, can decide what accesses to give, based on the identity of the entity
- It is possible that the two are merged

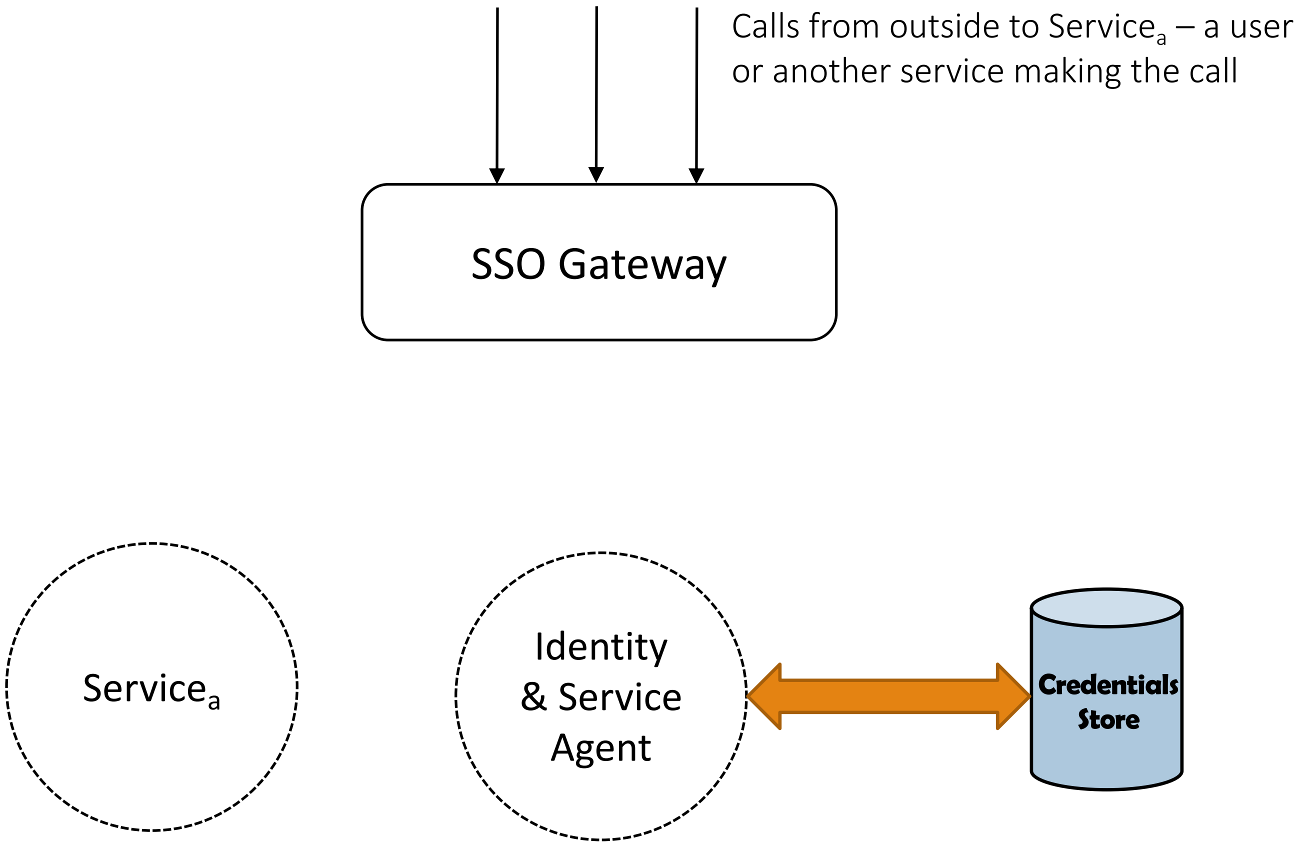
Implementing single sign-on

- Before attempting to implement your own SSO protocols, do take a look at **SAML** [19] and **OpenId connect** [20]

Using the SSO Gateway

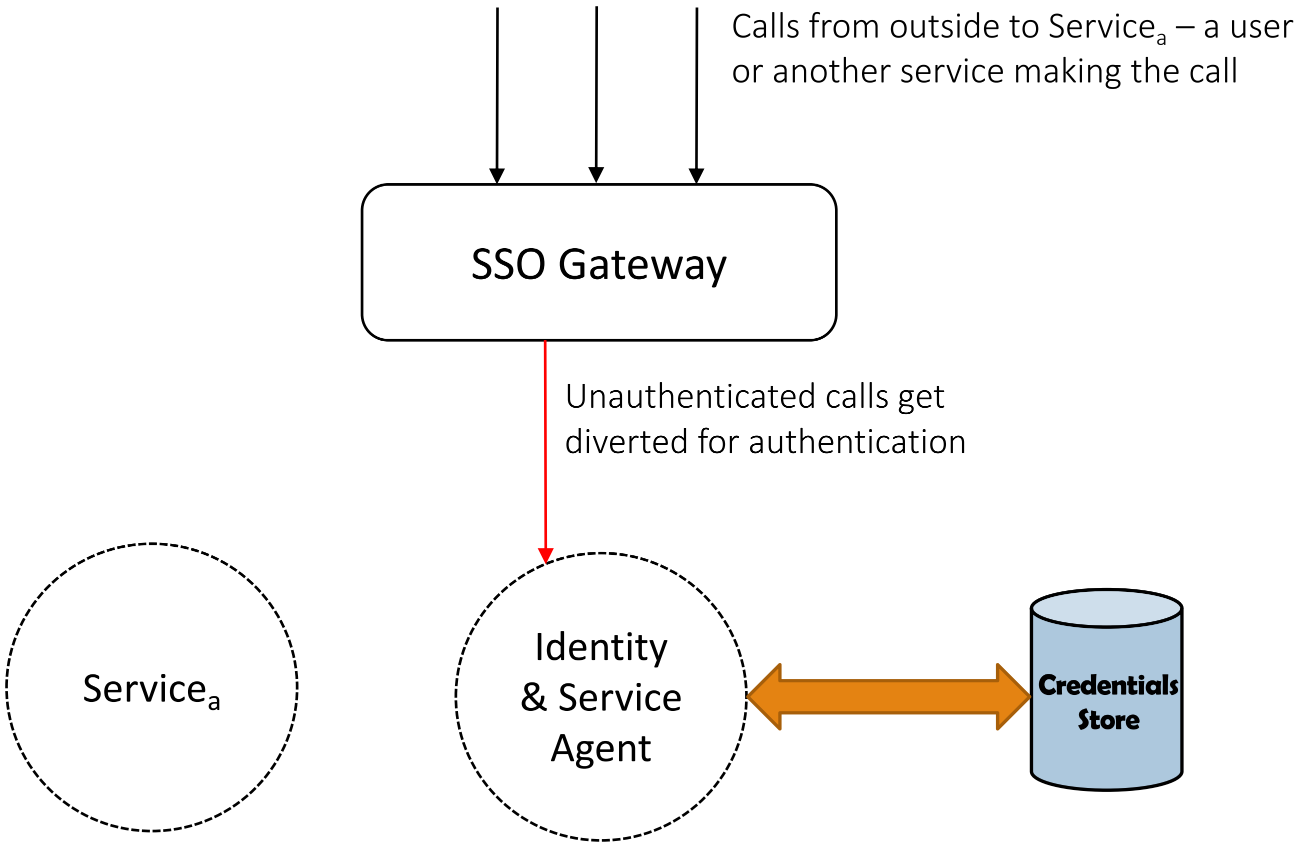


Using the SSO Gateway

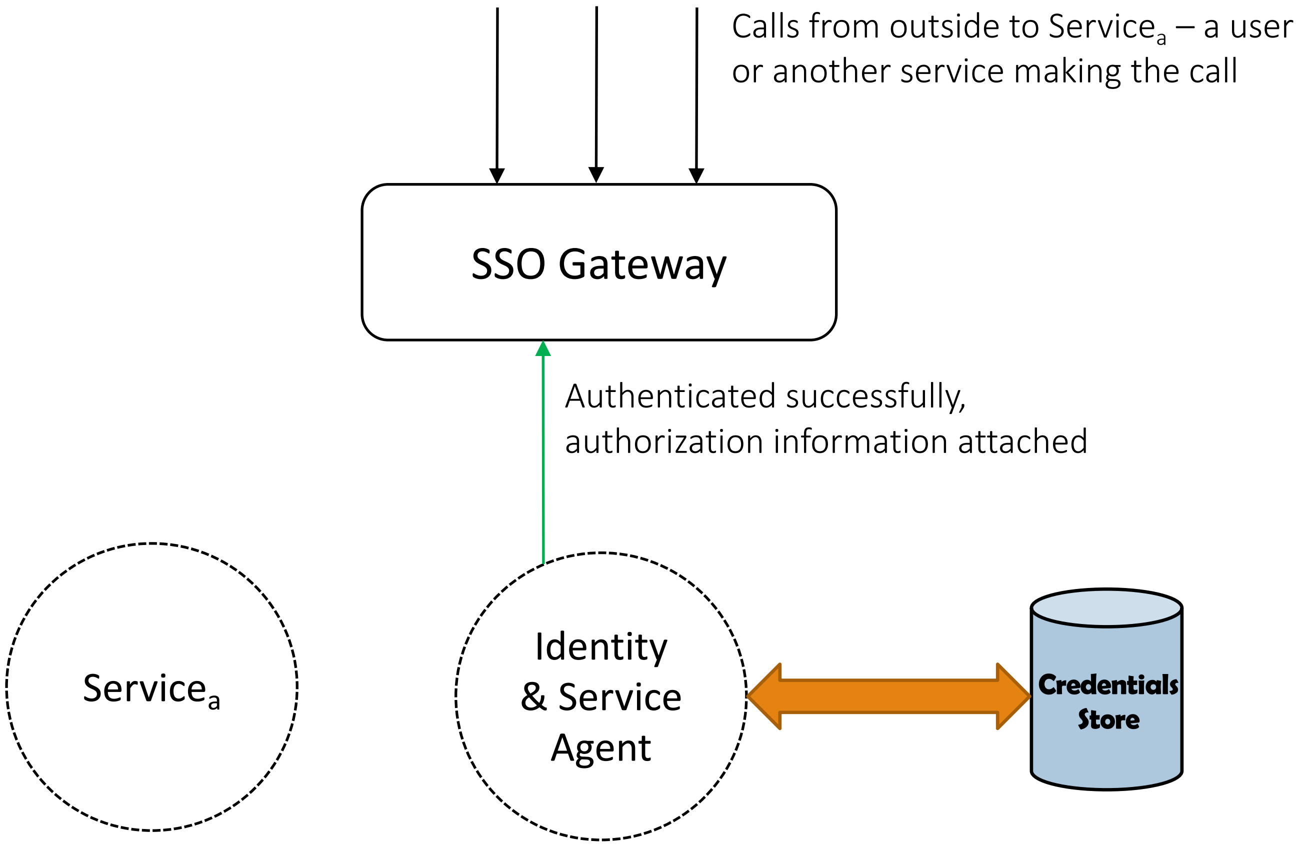


Calls from outside to Service$_a$ – a user or another service making the call

Using the SSO Gateway

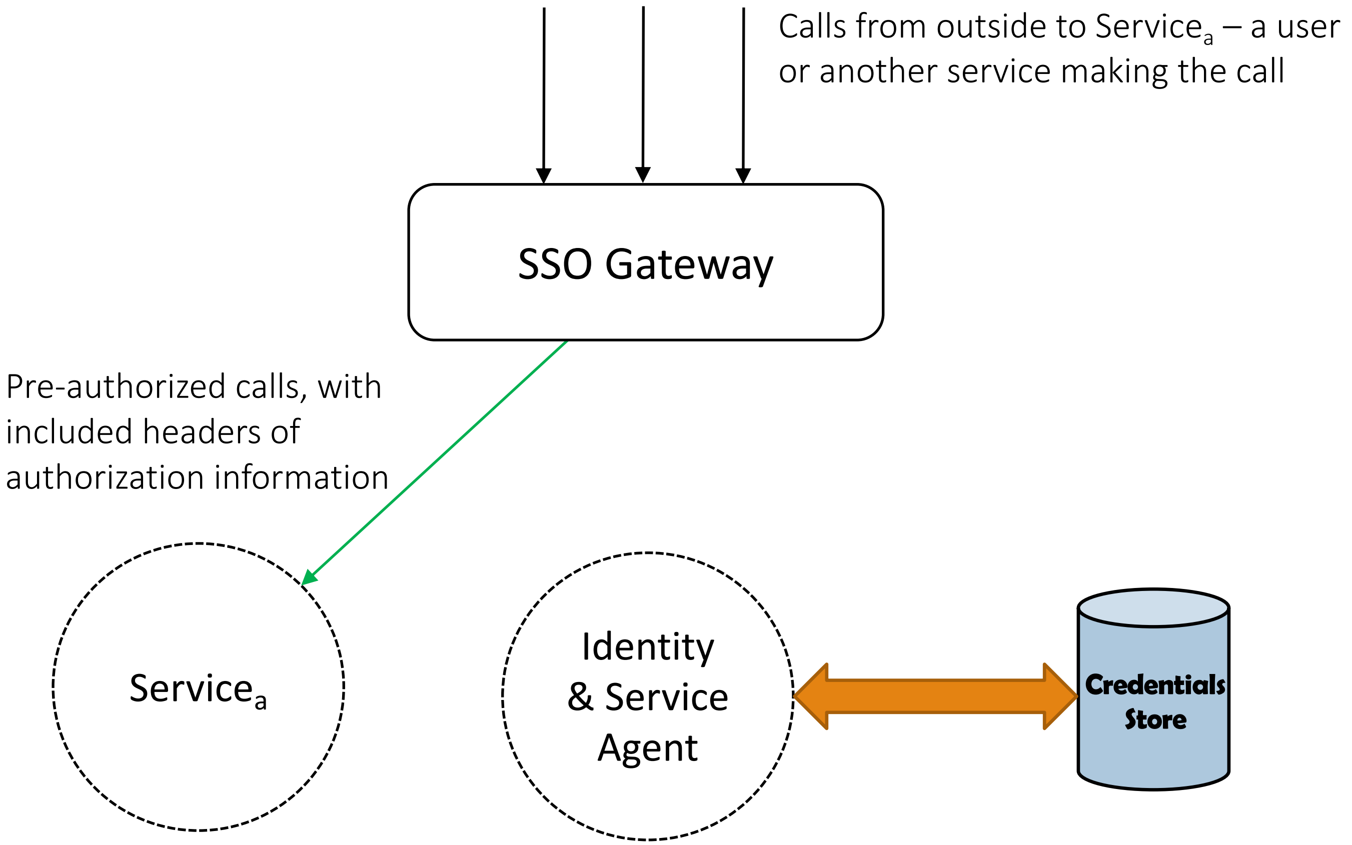

Calls from outside to Service$_a$ – a user or another service making the call
SSO Gateway
Unauthenticated calls get diverted for authentication
Service$_a$
Identity & Service Agent
Credentials Store

Using the SSO Gateway



Calls from outside to Service$_a$ – a user or another service making the call
SSO Gateway
Authenticated successfully, authorization information attached
Service$_a$
Identity & Service Agent
Credentials Store

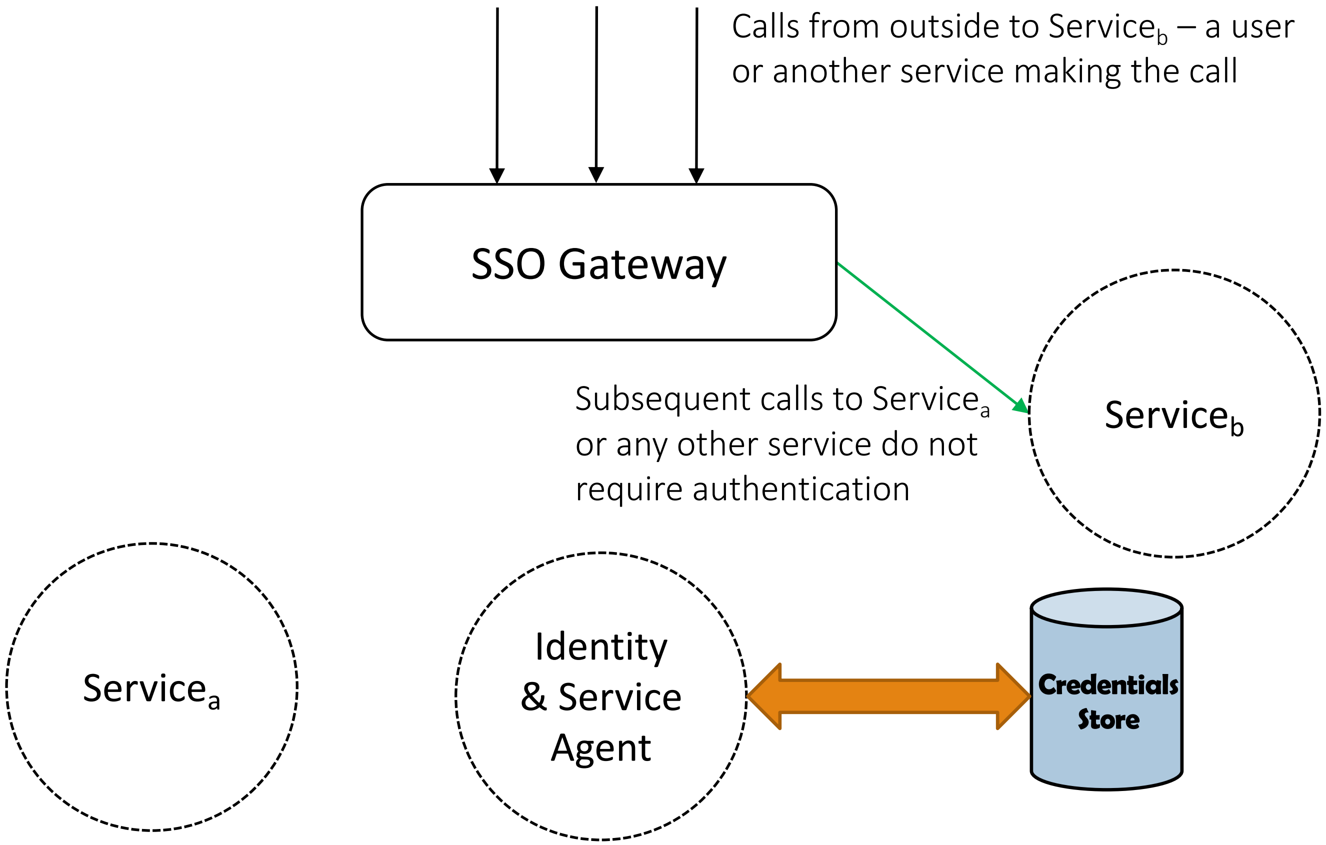
Using the SSO Gateway



Calls from outside to Service$_a$ – a user or another service making the call

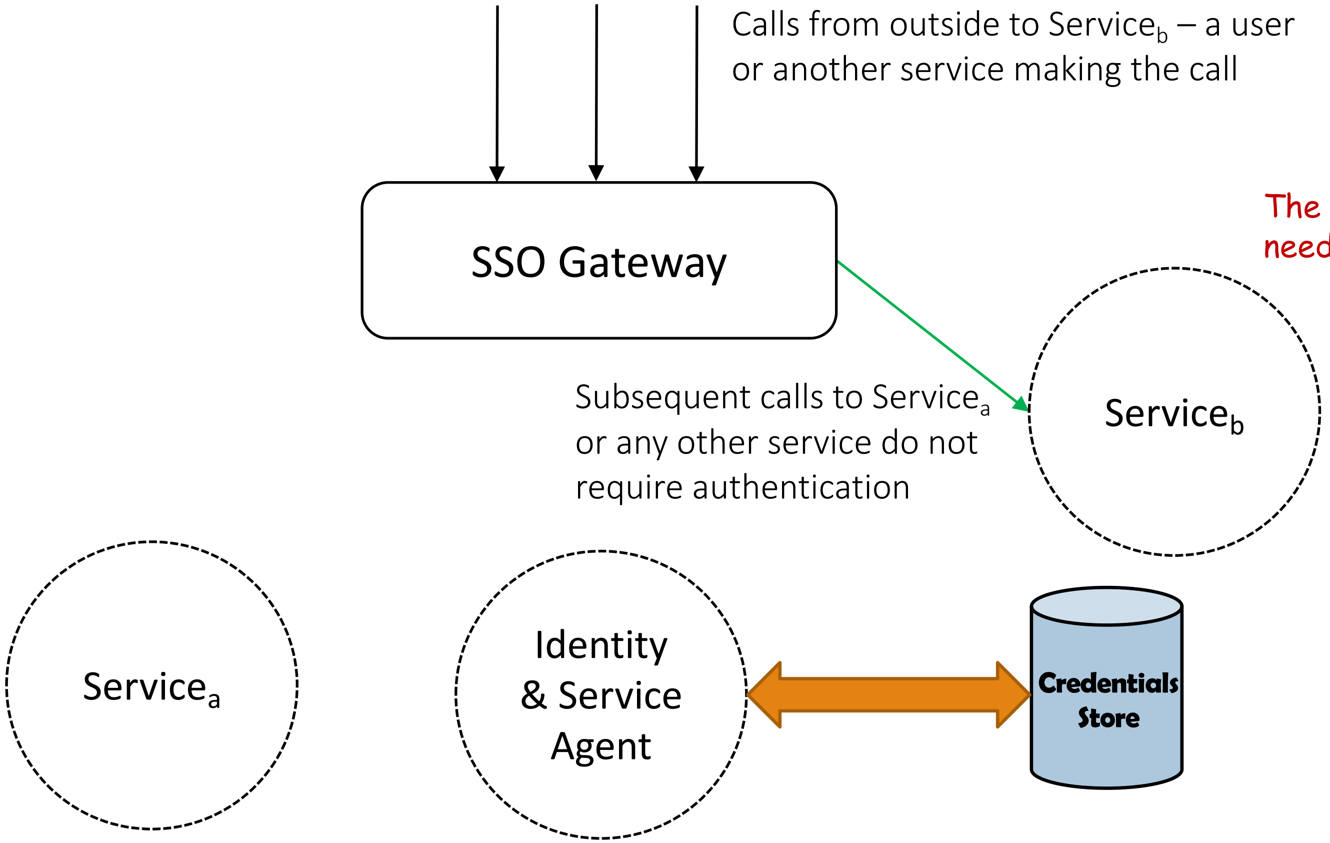
Pre-authorized calls, with included headers of authorization information

Using the SSO Gateway



Calls from outside to Service$_b$ – a user or another service making the call
SSO Gateway
Service$_b$
Subsequent calls to Service$_a$ or any other service do not require authentication
Service$_a$
Identity & Service Agent
Credentials Store

Using the SSO Gateway



Calls from outside to Service$_b$ – a user or another service making the call

SSO Gateway

Subsequent calls to Service$_a$ or any other service do not require authentication

Service$_b$

Service$_a$

Identity & Service Agent

Credentials Store

The SSO Gateway can decide which calls need authentication, and which doesn't

Service Authentication

How can the services be authenticated?

- Just like user passwords, services can also send *shared secrets* for authenticating themselves
- The communication, in both cases, need to be secure so that the secret is not leaked

Allowing everything within application boundary

- What if the invoker is a service within the application boundary?
- One approach is to *trust* other services in the applications, and provide them whatever access they seek, like returning the balance of a user, on providing the Account Number
- However, a badly written service, can expose the whole system in this case
- For instance, a malicious user may change the input parameters of the request, to put someone else's Account Number, and without a check, the call is passed downstream

Use SAML or OpenId connect for services too

- Services can be provided their own credentials, which they can use to authenticate themselves

Securing the transit – HTTPS

A secret shared over a public medium doesn't remain a secret

- Protocols like HTTP send data in plaintext, which can be easily read and spoofed
- HTTPS provides a layer of security to ensure that the communication between the parties is encrypted

Public Keys and Client Certificates

- Any involvement of Public-key cryptography brings with it issues like Certificate Management
- The verification and management of these certificates could be another added burden on the system
- Another related issue is to effectively manage events such as revocation of a certificate

HMAC over HTTP

- HTTPS can provide both privacy as well as integrity, but has its own overhead of encryption/decryption
- If all we want is integrity, we can use a lightweight solution of HMACS over HTTP
- The message to be sent is hashed, and encrypted with a pre-shared Key between the two entities
- The encrypted hash is sent along with the message, which can be verified at the receiver

API Keys

Some times, just authentication and authorization may not be enough

- We may need to put constraints such as *rate limiting*
- For example, we may disallow too much traffic from one service or user, so that others get a fair chance to get served

Pre-shared secrets

- An API key is a pre-shared secret between a service, and its consumers
- The API key thus allows a service to identify who is making a call
- It can then be used for finer grain operations such maintaining an access log
- It can also be used to block accesses to malicious entities

Securing stored data

Store only what's required

- The best way to keep data secured, is not to keep it at all
- If you require access to only certain information of the user, while rest is kept for *namesake*, get rid of it, to avoid leaking it in any attack

Make the data *less interesting*

- Some times, even if data does get leaked, it may not be useful for any attacks
- For example, if a *Recommendation Service* aggregates data like age-range for exact age/birthday and PIN Code for full address, then we only need to store these for a user, and not the detailed information

Choose what to encrypt

- Not all the data in your system may be critical, choose wisely what needs encryption
- Instead of encrypting entire databases, choose important tables to encrypt

Scaling and handling Failures

So we’ve built microservices

So we’ve built microservices

and we’ve deployed them too…

So we’ve built microservices

and we’ve deployed them too…

but they need to scale

So we’ve built microservices

and we’ve deployed them too…

but they need to scale

and they can fail !!

“Anything that can go wrong, will go wrong.”

Murphy’s Law ‘1952

Building for Failure

Failure is unavoidable

- An application deployed over internet should expect failures
- Nobody owns or control the internet, any link can go down, arbitrarily

MTTF vs MTTR

- Many enterprises try to increase the *Mean Time To Failure* (MTTF) of their applications
- They put extra resources and efforts to build applications that *don't fail*
- With microservices, it might be impossible to build an application that never fails
- We should rather expect them, and devise strategies to overcome or handle them gracefully
- In essence, instead of putting a lot of efforts to increase MTTF, try putting efforts to decrease MTTR or *Mean Time To Repair*

Patterns to handle Failure

In his book *Release It!*, **Michael Nygard** [10] has taken abstractions from the real world, which can be used in the software arena.

- He suggests some novel ways to detect and handle failures in a distributed system

Timeouts

- The most simple yet useful pattern is to use *timeouts*
- All out-of-process calls shall have timeouts associated with them
- If the request is not fulfilled within a specified time, consider the downstream service to be down
- Timeouts shall be selected wisely – too short, and we may falsely assume a service to be down; too large and we may slow down the whole system

Circuit Breakers

Use *Circuit Breakers* between out-of-process calls

- Every service is expected to perform according to some performance metrics
- One such metrics could be number of responses given per unit time
- Another metrics could be average response time to serve a request
- If a downstream service is *not performing* as per expectations, it may be *too burdened*
- A Circuit-breaker monitors the service for any performance parameter, and *blows* the breaker to make the invoking service treat the downstream service as down
- The invoking service get a *fast failure* response, which it can handle, or push upstream
- Provides a breathing space for the downstream process to recover from pending load
- A periodic *health checkup* can be performed to see if the downstream process is *healthy*, in which case, the breaker is reset

Circuit Breakers

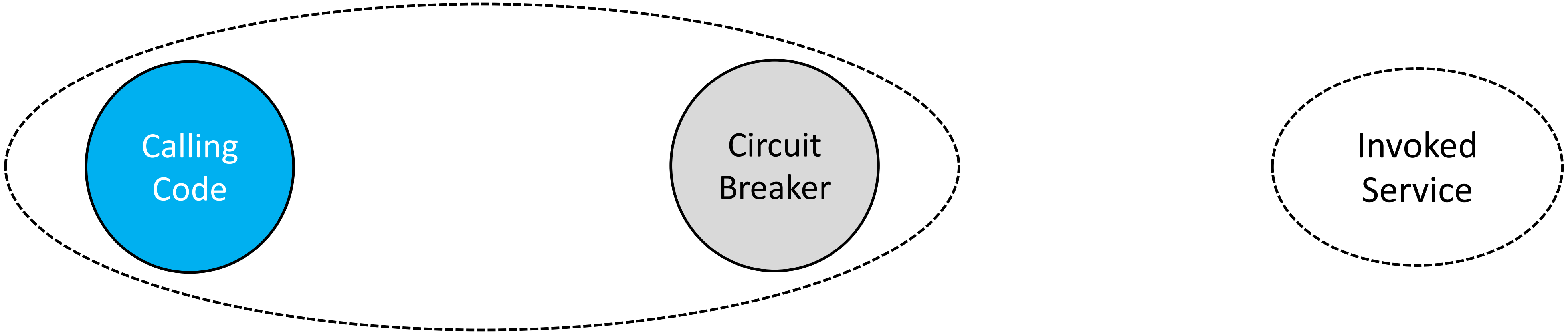
Use *Circuit Breakers* between out-of-process calls

- Circuit Breakers can also be used for Maintenance purposes
- A team which needs to perform a maintenance on its service, or are planning to deploy a newer version, can *blow all the breakers* connecting to them manually
- Once the service is back online, the breakers can be reset

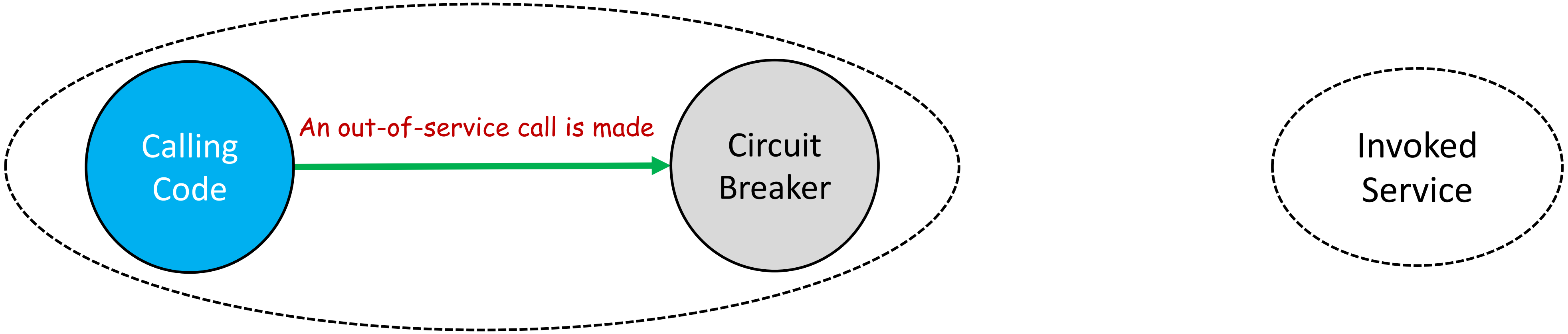
Tools to implement Circuit Breakers

- Netflix's **Hystrix** [25] library is for circuit breaker implementations in JVM
- **Polly** [26] is a library which allows you to define circuit breaker policies in .Net
- **Stoplight** [27] is a recent Circuit Breaker implementation for Ruby

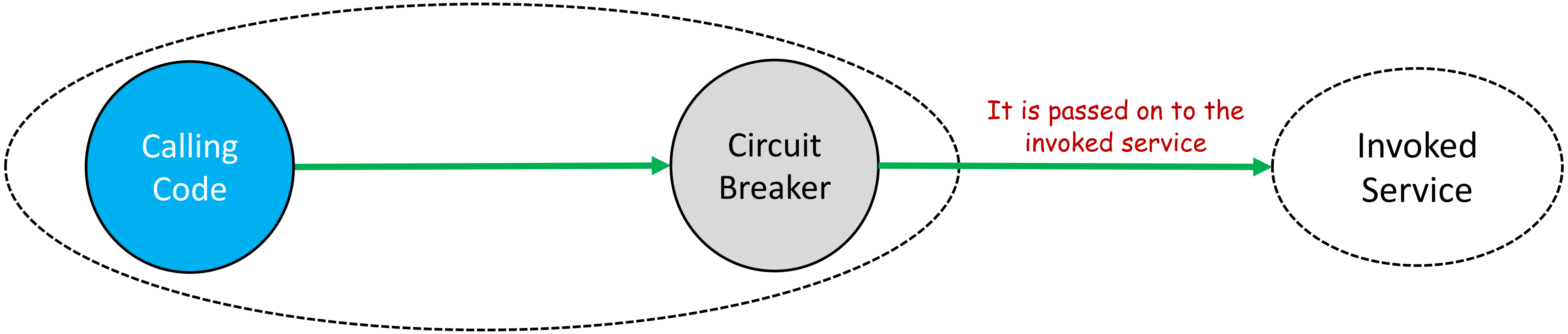
Circuit Breakers: Sample Implementation



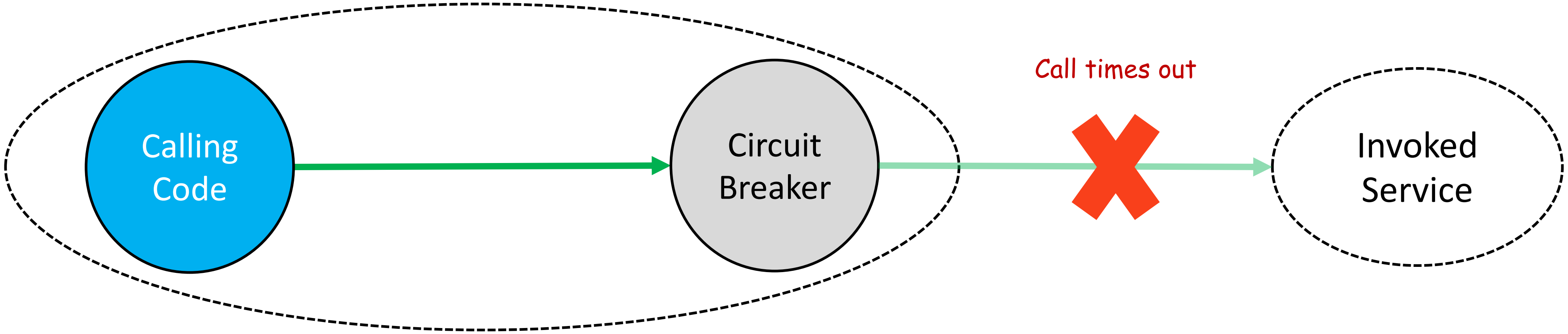
Circuit Breakers: Sample Implementation



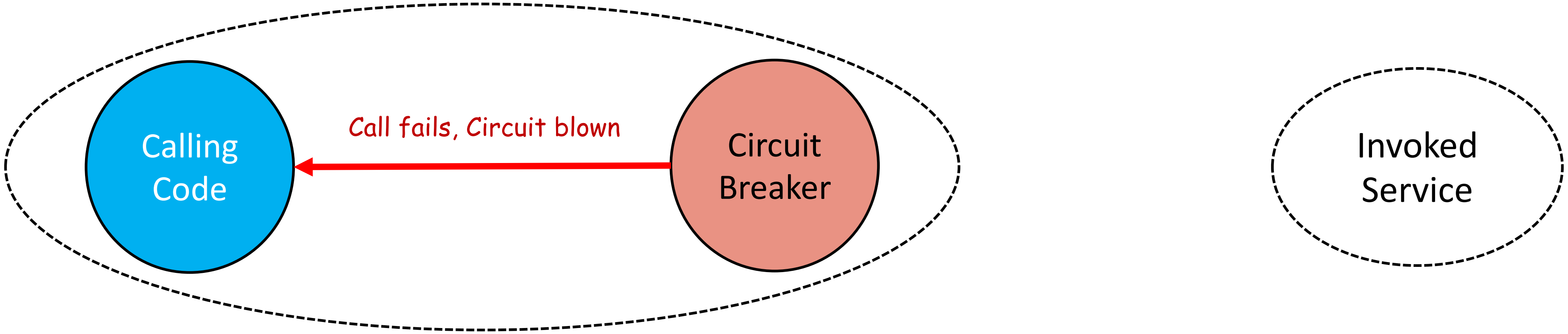
Circuit Breakers: Sample Implementation



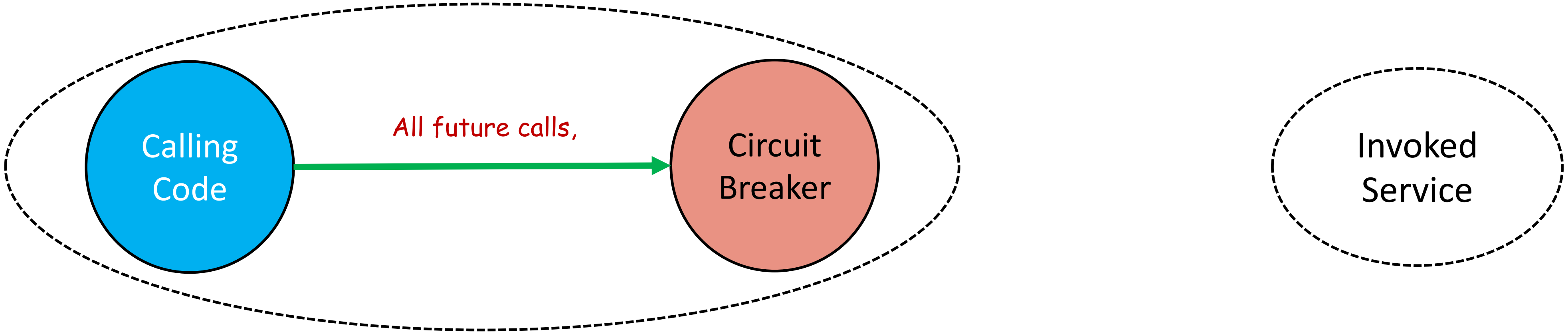
Circuit Breakers: Sample Implementation



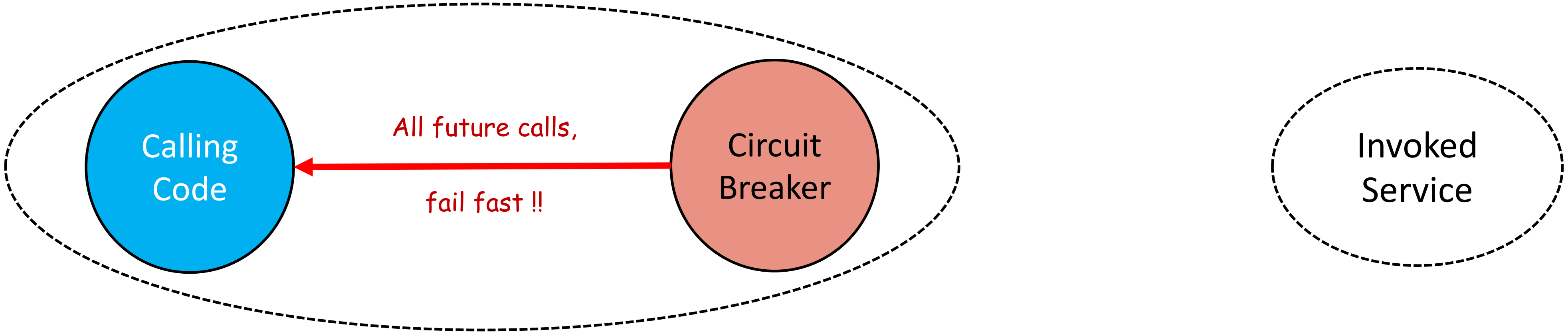
Circuit Breakers: Sample Implementation



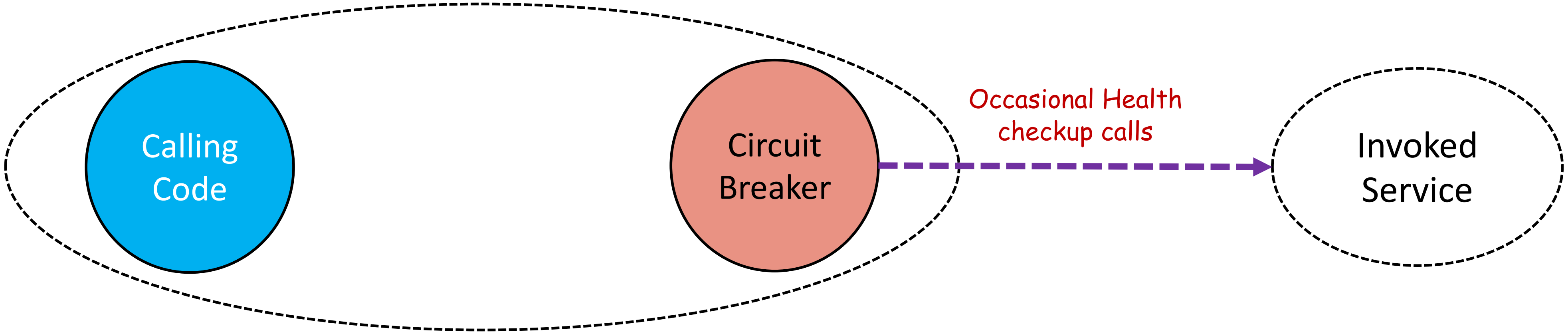
Circuit Breakers: Sample Implementation



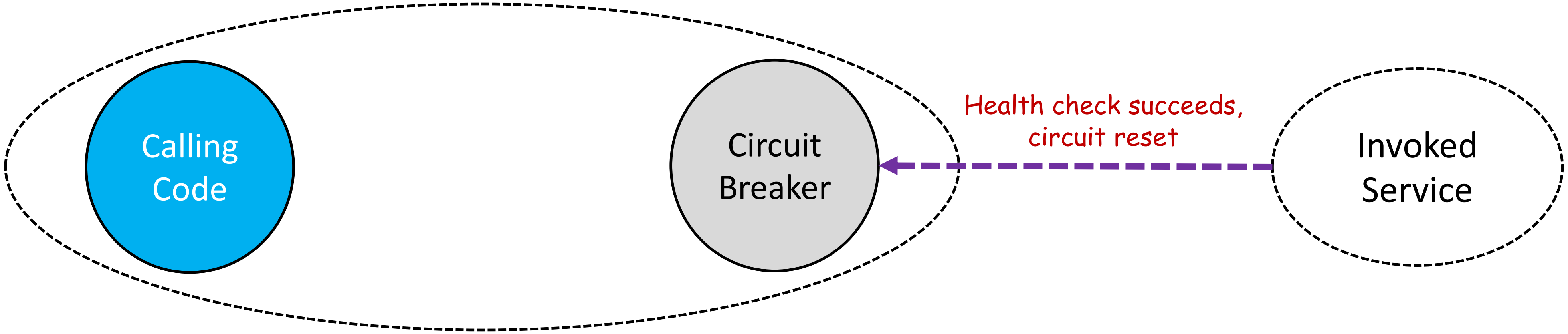
Circuit Breakers: Sample Implementation



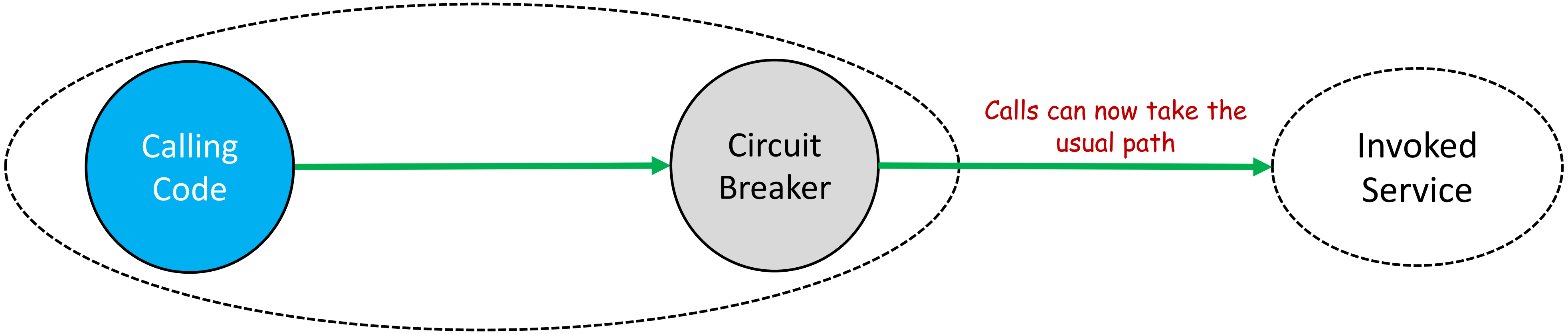
Circuit Breakers: Sample Implementation



Circuit Breakers: Sample Implementation



Circuit Breakers: Sample Implementation



Bulkheads

The concept of bulkheads come from the shipping ecosystem

- If a part of the ship gets damaged, sealing it off from the rest of the ship, can save the whole ship from drowning

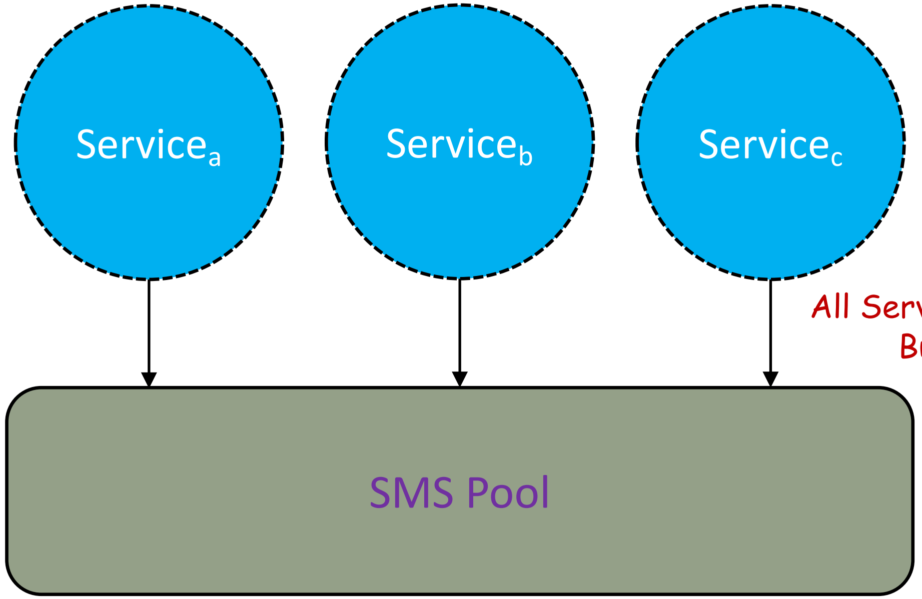
If we are sure that a process is misbehaving, or in poor health, seal it off from others

- Think about a scenario where all services use worker threads from a common pool to do their jobs
- One misbehaving service can exhaust the whole pool, starving others too, bringing the system to a halt
- We can instead, put a pool quota for all services or a separate pool itself for everyone
- The erring service will starve after exhausting its pool, yet others can continue to work

In some systems, a service may choose to go in isolation all by itself

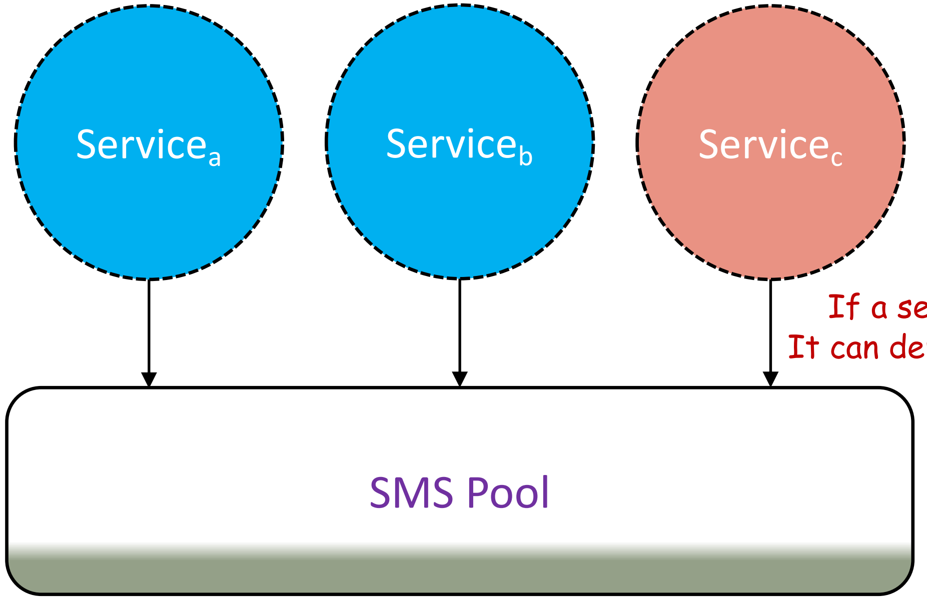
- A service may decide not to honor any new requests, until it *gets well*
- The pending requests may go to a queue or may be rejected outright, depending on implementation

Bulkheads: Sample Implementation



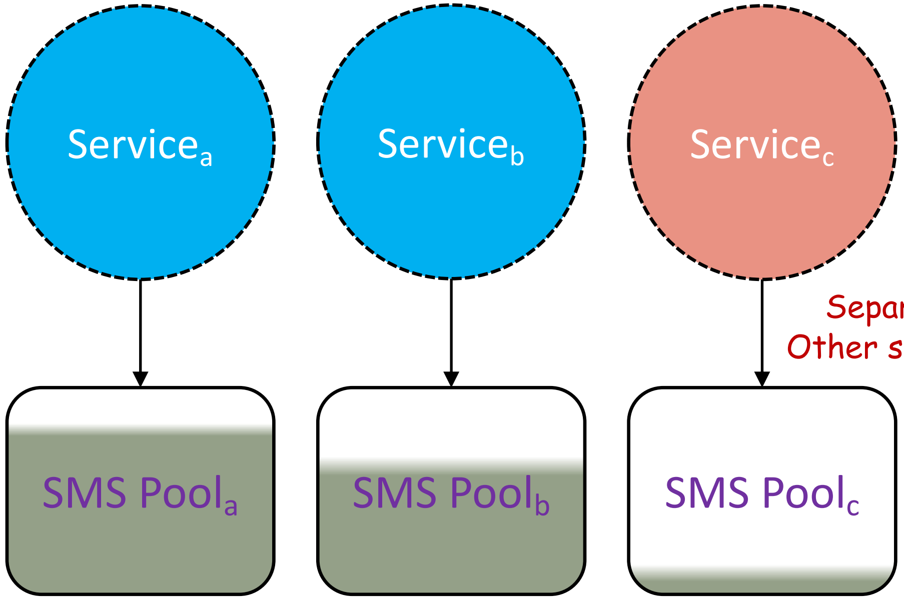
All Services use a common
Bulk SMS pool

Bulkheads: Sample Implementation



If a service goes rogue,
It can deplete the whole pool

Bulkheads: Sample Implementation



Separate the pools, so that
Other services are not affected

Scaling – why should we bother?

There are two common scenarios when we need *more of something*, i.e. we need to operate with a large number of instances of one or more services

We are worried that things will fail

- In a distributed systems, things can, and will fail
- The more the merrier

We want to avoid performance degradation

- What if there is a spike in load?
- We need to be able to respond to a surge in requests, without compromising performance

How can we scale microservices?

Buy bigger boxes (or hosts)

- The simplest way to scale
- Vertical Scaling – buy a machine or host, with higher configuration
- Works well if multiple microservices are deployed on the same host
- One or more cores can be dedicated to certain microservices

Not really scalable for large systems

- There can be cases where a server with higher configurations, costs more than a sum of smaller machines adding up to the same resources
- Doesn’t work well if we deploy one instance per host, and the services can not harness more cores

How can we scale microservices?

Spreading risks geographically

- To avoid failures resulting in a complete halt of the system, keep the hosts geographically separated
- Generally possible with commercial virtualization platforms, like AWS
- Put your services in *different regions* to decrease the probability of a total outage due to network failure

Costly affair

- Building and managing your own Data Centers separated geographically would be a costly affair
- Using Virtual Machines as hosts, and buying them from a Cloud Service Provider could be much cheaper

How can we scale microservices?

Load Balancing

- Instead of putting heavy loads on some instances, and light loads on others, use Load Balancers to balance loads between the services
- Load Balancers can also perform other important tasks such as state information forwarding, SSL Termination, Caching etc.
- The services themselves can then reside in a different subnet (or a Virtual Private Cloud as they are known in AWS), providing a level of security to services from outside world

How can we scale microservices?

Worker based systems

- Use a pool of resources from which Services pickup *workers* to get their jobs done
- A spike in load can be handled by putting more workers in the pool to handle the requests

How can we scale microservices?

Scaling Databases

- Just scaling the services may not work
- Database accesses may still be a bottleneck

Use *Read Replicas* and *Caching* to scale for Reads

- If the writes are much less frequent than reads, this model can scale fairly well

Use *Shadowing* or *Horizontal Portioning*, for Writes

- Divide the rows of the database among multiple instances
- A Write Request may now be mapped to a subset of overall database
- Problem: Range queries and update to multiple rows will be inefficient
- Mongo DB uses Map/Reduce jobs to answer such queries
- Another issue is adding a new *shard* (partition) to the system without going offline
- Cassandra can do this fairly well

Effective Caching

Different levels of Caching

- Caching helps improve performance, and reduce loads on services
- Effective caching can reduce redundant calls to servers
- Caching can be done at multiple levels

Client Level

- The client caches the response it gets from the server
- The client decides what to cache, and what not to
- Problem: difficult to roll out new changes
- The clients may wish to use a cached version of the resource, rather than fetching the new, updated one

Effective Caching

Different levels of Caching

- Caching helps improve performance, and reduce loads on services
- Effective caching can reduce redundant calls to servers
- Caching can be done at multiple levels

Proxy Level

- A proxy that sits between Server and Client can cache two-way traffic
- Both Client as well as server are oblivious to it
- Some communication protocols like HTTP, provide ample amount of header information, for any in between tool to cache certain responses
- For example, the `Expires` header can provide information about how long a resource is not supposed to change
- Reverse Proxies like **Squid** [21] or **Varnish** [22] can use such information to cache a lot of responses

Effective Caching

Different levels of Caching

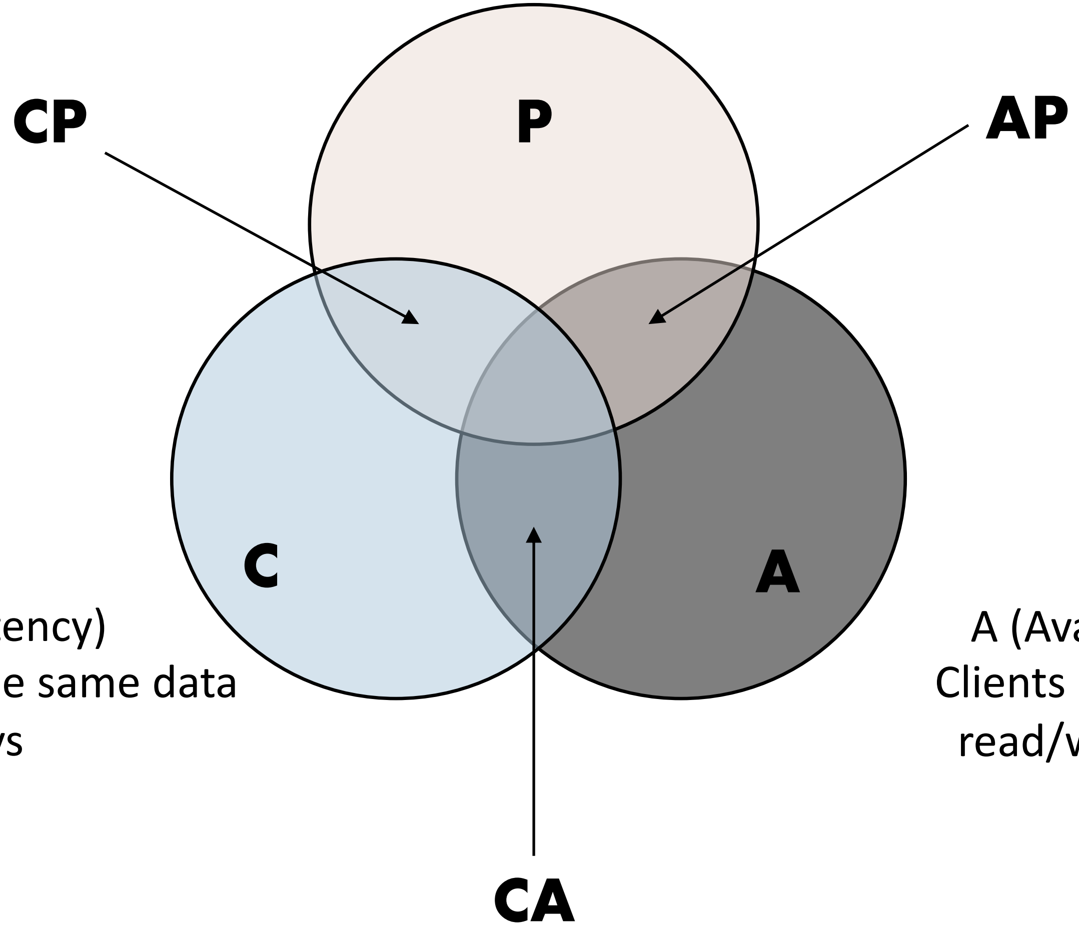
- Caching helps improve performance, and reduce loads on services
- Effective caching can reduce redundant calls to servers
- Caching can be done at multiple levels

Server Level

- Server takes responsibility of all the caching, client doesn’t have to worry about caching at all
- If the clients can be of varying types (services, browsers, native apps etc.), server side caching may be of significant advantage
- Tools like **Memcached** [24] and **Redis** [23] can be used for such purposes

The CAP Theorem

P (Partition Tolerance)
System is Partitioned



C (Consistency)
All Clients see the same data always

A (Availability)
Clients can always read/write data

AP vs CP

Since we are talking in terms of microservices, leaving out *P* is not really an option

- Which leaves us two modes of operations to choose from – AP or CP?

Implementing consistency is generally not easy in a distributed environment

- Irrespective of whether we want availability or not
- Implementing consistency may involve Distributed Commits, which have their own issues
- Still, some operations, like deduction of money from a bank account, mandates consistency

Choosing *eventual consistency* where it can work

- In case providing stale results to some reads may not make a difference, the model of eventual consistency can be bought
- All instances of a database sync themselves up “eventually” after a write

Services may not have to go either way all the times

- Some operations can be chosen to have consistency, while others can choose eventual consistency

Service Discovery and Configuration

How do our services find each other?

- We need a mechanism where a service can announce its presence and others can look it up

General Service Discovery scenario

- A service records its presence and other information, such as IP and ports to access it, on a common platform
- Any consumer service does a look up on this platform, to find the details to contact the service

DNS

- Easiest way to implement service discovery
- Services put their IPs as DNS entries at a central location for lookups
- Works well if there are single instances of each service
- Problem: hard to update DNS entries, and handling of multiple instances of same service
- Probable Solution: One service instance acts like a load balancer for others, the DNS entry can then point to this instance, and it may delegate the same to others

Service Discovery and Configuration

How do our services find each other?

- We need a mechanism where a service can announce its presence and others can look it up

Zookeeper

- Zookeeper [28] was originally developed as a part of Hadoop project
- Has a wide range of features like data synchronization, leader election, message queue as well as a naming service
- Provides a hierarchical namespace to which clients can add, modify and query nodes
- Clients can also add watches to nodes, and get intimated if that node changes (e.g. a change in IP)
- Is rather old now, but tried and tested

Service Discovery and Configuration

Consul

- Consul [29] provides an HTTP interface via RESTful API to register or query services
- It allows to insert health statistics too for a particular service
- The use of REST makes it easily pluggable with a lot of current systems
- Comes with an out-of-box DNS server too
- Is new, but seems to be getting popular

Eureka

- Eureka [30] is an Open Source service registry solution by Netflix, primarily for services on AWS
- Unlike Zookeeper or Consul, Eureka doesn’t attempt to be a general-purpose configuration store as well
- Also comes with basic load balancing capabilities
- Provides RESTful API as well as a Java Client for usage
- The Java Client can also do additional operations such as health checking of instances

Issues to ponder !

Microservices: fishing in troubled waters

How to analyze the big picture without building up all the smaller systems?

- Since we decouple the whole process of building the application, the actual application is not available until all parts or at least some parts are ready, and deployed to communicate

Service Size?

- One of the most asked questions in the field of microservices, is how small is a microservice?
- Is there a concept of *macroservice* then which we can call an anti-pattern? If yes, how can we judge if the service is a macroservice?

Isn’t it just *fine grained SOA*?

- Some people argue that microservices are nothing new, but a finer grained version of SOA
- Are we sure microservices won’t fade away the same way SOA did?

How to find Service Boundaries that match Bounded Contexts?

- It may be easier said than done
- What if a model is equally important to two or more Bounded Contexts? Who’ll be the owner?

Microservices: fishing in troubled waters

How to test microservices effectively?

- One of the major reasons monoliths take more time to market, is because of the exhaustive test suites that must be completed before a release
- With microservices, we put more trust on developers to be able to provide inputs on how to test their services, or rely on CDCs, expecting that they'll cover any problems that may creep in
- Also, how do we test the deployment environment? In most cases, it is not possible to replicate the massive production environment, on laptops or desktops

Lack of tool support

- Despite a number of tools that exist for various aspects about microservices development, we are still in need of more tools
- To be precise, tools for automating deployment and management of infrastructure, are still fairly naïve, and leave lot to desire

A good understanding of DevOps approach may be essential to succeed with Microservices

Wrapping up !!

Summary

Microservices Architecture

- is an Architectural Style
- meant to build scalable, resilient, easily updateable applications
- using small teams
- by building small, autonomous microservices,
- which are loosely coupled with smarts in the end-points, not in the network

However,

- they bring with them issues associated with
- managing distributes resources,
- and sub-optimal testing processes,
- along with duplication of code and effort at times

Conclusion

We discussed various aspects of the Microservices Architecture Style

- Starting from their **need** in the current scenario
- We then discussed the **trade-off** that lie in store if we choose to pick it for our development
- Followed by how we may **split a monolith** into a set of small autonomous services
- We also discussed a classification of tests and some **testing** strategies fit for microservices
- We skimmed through the various **monitoring** issues we may encounter while operating microservices
- We talked about how to go about implementing **security** in our applications built using microservices
- We then discussed how microservices **operate on scale**, and how they **handle failures**
- We concluded with a note on various **challenges** a practitioner may face while using microservices

Thanks !

References

[1] Release engineering and push karma, **Chuck Rossi**, https://www.facebook.com/note.php?note_id=10150660826788920

[2] Micro services, what even are they?, **Jon Eaves**, http://techblog.realestate.com.au/micro-services-what-even-are-they/

[3] Building Microservices – Designing Fine Grained Systems, **Sam Newman**, O'Reilly

[4] Working Effectively with Legacy Code, **Michael Feathers**, Prentice Hall

[5] Succeeding with Agile: Software Development Using Scrum, **Mike Cohn.**, Addison-Wesley

[6] Continuous Integration, Build Pipelines and Continuous Deployment, **Christopher Read**, http://www.slideshare.net/ChristopherRead/continuous-integration-build-pipelines-and-continuous-deployment

[7] Jenkins: An extensible open source continuous integration server, https://jenkins-ci.org/

[8] Bamboo: Continuous Integration & Build Server, https://www.atlassian.com/software/bamboo

References

[9] Packer: A tool for creating machine and container images for multiple platforms from a single source configuration, https://www.packer.io/

[10] Release It!: Design and Deploy Production-Ready Software, **Michael T. Nygard**, Pragmatic Programmers

[11] Pact, RealEstate.com.au, https://github.com/realestate-com-au/pact

[12] ClusterSSH, Cluster administration tool, http://sourceforge.net/projects/clusterssh/

[13] Pdsh, High-performance parallel remote shell utility, https://code.google.com/p/pdsh/

[14] Nagios, IT Infrastructure monitoring tool, https://www.nagios.org/

[15] Logstash, “Collect, Parse, Transform Logs”, https://www.elastic.co/products/logstash

[16] Kibana, “Explore, Visualize, Discover Data”, https://www.elastic.co/products/kibana

[17] Graphite, Scalable Realtime Graphing, http://graphite.wikidot.com/

References

[18] Zipkin: A distributed tracing system, http://twitter.github.io/zipkin/

[19] Security Assertion Markup Language (SAML) v2.0, Organization for the Advancement of Structured Information Standards (OASIS), http://saml.xml.org/

[20] OpenID Connect: A Simple Identity layer on top of OAuth 2.0, http://openid.net/

[21] Squid: A caching proxy for web, http://www.squid-cache.org/

[22] Varnish: An HTTP accelerator, https://www.varnish-cache.org/

[23] Redis: An advanced Key-value cache and store, http://redis.io

[24] Memcached: A distributed memory object caching system, http://memcached.org/

[25] Hystrix: A latency and fault tolerance library, https://github.com/Netflix/Hystrix

[26] Polly: A .Net Library for transient exceptional handling, https://github.com/michael-wolfenden/Polly

References

[27] Spotlight: Traffic control for code, https://github.com/orgsync/stoplight

[28] Zookeeper: Configuration tool for group services, https://zookeeper.apache.org

[29] Consul: Service discovery and configuration made easy, https://www.consul.io

[30] Eureka: AWS Service registry for resilient mid-tier load balancing and failover, https://github.com/Netflix/eureka