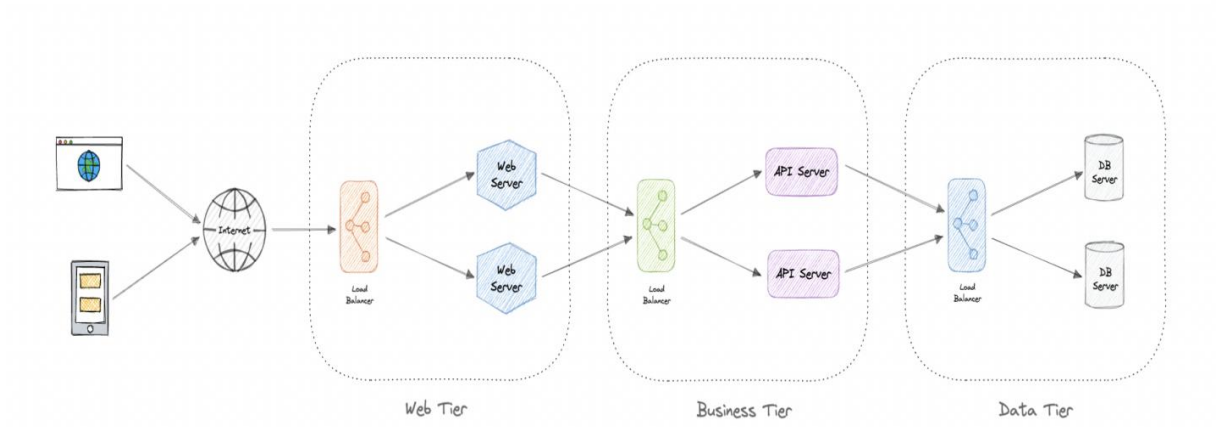
ASYNCHRONOUS SYSTEMS, MESSAGING MONOLITHS AND MICROSERVICES

N-tier architecture

N-tier architecture divides an application into logical layers and physical tiers. Layers are a way to separate responsibilities and manage dependencies. Each layer has a specific responsibility. A higher layer can use services in a lower layer, but not the other way around.



Tiers are physically separated, running on separate machines. A tier can call to another tier directly, or use asynchronous messaging. Although each layer might be hosted in its own tier, that's not required. Several layers might be hosted on the same tier. Physically separating the tiers improves scalability and resiliency and adds latency from the additional network communication.

An N-tier architecture can be of two types:

- In a closed layer architecture, a layer can only call the next layer immediately down.
- In an open layer architecture, a layer can call any of the layers below it.

A closed-layer architecture limits the dependencies between layers. However, it might create unnecessary network traffic, if one layer simply passes requests along to the next layer.

Types of N-Tier architectures

Let's look at some examples of N-Tier architecture:

3-Tier architecture

3-Tier is widely used and consists of the following different layers:

- **Presentation layer**: Handles user interactions with the application.
- **Business Logic layer**: Accepts the data from the application layer, validates it as per business logic and passes it to the data layer.
- **Data Access layer**: Receives the data from the business layer and performs the necessary operation on the database.

2-Tier architecture

In this architecture, the presentation layer runs on the client and communicates with a data store. There is no business logic layer or immediate layer between client and server.

Single Tier or 1-Tier architecture

It is the simplest one as it is equivalent to running the application on a personal computer. All of the required components for an application to run are on a single application or server.

Advantages

Here are some advantages of using N-tier architecture:

- Can improve availability.
- Better security as layers can behave like a firewall.

- Separate tiers allow us to scale them as needed.
- Improve maintenance as different people can manage different tiers.

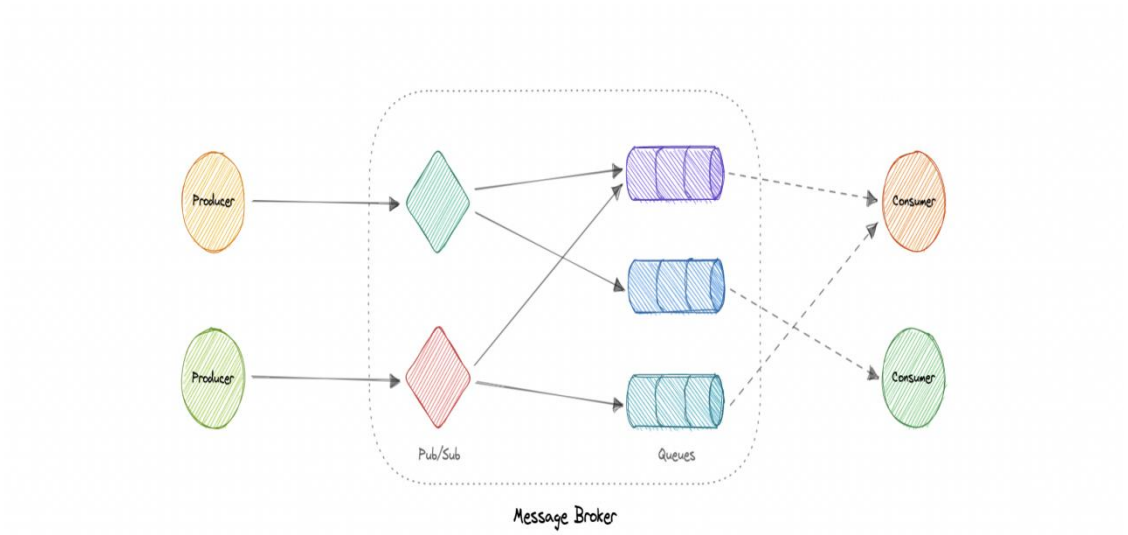
Disadvantages

Below are some disadvantages of N-tier architecture:

- Increased complexity of the system as a whole.
- Increased network latency as the number of tiers increases.
- Expensive as every tier will have its own hardware cost.
- Difficult to manage network security.

Message Brokers

A message broker is software that enables applications, systems, and services to communicate with each other and exchange information. The message broker does this by translating messages between formal messaging protocols. This allows interdependent services to "talk" with one another directly, even if they were written in different languages or implemented on different platforms.



Message brokers can validate, store, route, and deliver messages to the appropriate destinations. They serve as intermediaries between other applications, allowing senders to issue messages without knowing where the receivers are, whether or not they are active, or how many of them there are. This facilitates the decoupling of processes and services within systems.

Models

Message brokers offer two basic message distribution patterns or messaging styles:

- Point-to-Point messaging: This is the distribution pattern utilized in message queues with a one-to-one relationship between the message's sender and receiver.

- Publish-Subscribe messaging: In this message distribution pattern, often referred to as "pub/sub", the producer of each message publishes it to a topic, and multiple message consumers subscribe to topics from which they want to receive messages.

We will discuss these messaging patterns in detail in the later tutorials.

Message brokers vs. Event streaming

Message brokers can support two or more messaging patterns, including message queues and pub/sub, while event streaming platforms only offer pub/sub-style distribution patterns. Designed for use with high volumes of messages, event streaming platforms are readily scalable. They're capable of ordering streams of records into categories called topics and storing them for a predetermined amount of time. Unlike message brokers, however, event streaming platforms cannot guarantee message delivery or track which consumers have received the messages.

Event streaming platforms offer more scalability than message brokers but fewer features that ensure fault tolerance like message resending, as well as more limited message routing and queuing capabilities.

Message brokers vs. Enterprise Service Bus (ESB)

Enterprise Service Bus (ESB) infrastructure is complex and can be challenging to integrate and expensive to maintain. It's difficult to troubleshoot them when problems occur in production environments, they're not easy to scale, and updating is tedious.

Whereas message brokers are a "lightweight" alternative to ESBs that provide similar functionality, a mechanism for inter-service communication, at a lower cost.

They're well-suited for use in the microservices architectures that have become more prevalent as ESBs have fallen out of favor.

Examples

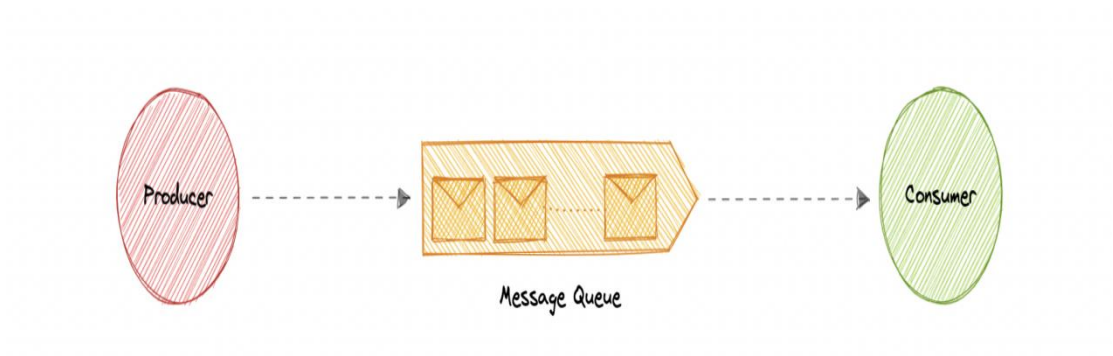
Here are some commonly used message brokers:

- NATS
- Apache Kafka
- RabbitMQ
- ActiveMQ

Message Queues

A message queue is a form of service-to-service communication that facilitates asynchronous communication. It asynchronously receives messages from producers and sends them to consumers.

Queues are used to effectively manage requests in large-scale distributed systems. In small systems with minimal processing loads and small databases, writes can be predictably fast. However, in more complex and large systems writes can take an almost non-deterministic amount of time.



Working

Messages are stored in the queue until they are processed and deleted. Each message is processed only once by a single consumer. Here's how it works:

- A producer publishes a job to the queue, then notifies the user of the job status.
- A consumer picks up the job from the queue, processes it, then signals that the job is complete.

Advantages

Let's discuss some advantages of using a message queue:

- **Scalability**: Message queues make it possible to scale precisely where we need to. When workloads peak, multiple instances of our application can add all requests to the queue without the risk of collision.

- **Decoupling**: Message queues remove dependencies between components and significantly simplify the implementation of decoupled applications.
- **Performance**: Message queues enable asynchronous communication, which means that the endpoints that are producing and consuming messages interact with the queue, not each other. Producers can add requests to the queue without waiting for them to be processed.
- **Reliability**: Queues make our data persistent, and reduce the errors that happen when different parts of our system go offline.

Features

Now, let's discuss some desired features of message queues:

Push or Pull Delivery

Most message queues provide both push and pull options for retrieving messages. Pull means continuously querying the queue for new messages. Push means that a consumer is notified when a message is available. We can also use long-polling to allow pulls to wait a specified amount of time for new messages to arrive.

FIFO (First-In-First-Out) Queues

In these queues, the oldest (or first) entry, sometimes called the "head" of the queue, is processed first.

Schedule or Delay Delivery

Many message queues support setting a specific delivery time for a message. If we need to have a common delay for all messages, we can set up a delay queue.

At-Least-Once Delivery

Message queues may store multiple copies of messages for redundancy and high availability, and resend messages in the event of communication failures or errors to ensure they are delivered at least once.

Exactly-Once Delivery

When duplicates can't be tolerated, FIFO (first-in-first-out) message queues will make sure that each message is delivered exactly once (and only once) by filtering out duplicates automatically.

Dead-letter Queues

A dead-letter queue is a queue to which other queues can send messages that can't be processed successfully. This makes it easy to set them aside for further inspection without blocking the queue processing or spending CPU cycles on a message that might never be consumed successfully.

Ordering

Most message queues provide best-effort ordering which ensures that messages are generally delivered in the same order as they're sent and that a message is delivered at least once.

Poison-pill Messages

Poison pills are special messages that can be received, but not processed. They are a mechanism used in order to signal a consumer to end its work so it is no longer waiting for new inputs, and are similar to closing a socket in a client/server model.

Security

Message queues will authenticate applications that try to access the queue, this allows us to encrypt messages over the network as well as in the queue itself.

Task Queues

Tasks queues receive tasks and their related data, run them, then deliver their results. They can support scheduling and can be used to run computationally intensive jobs in the background.

Backpressure

If queues start to grow significantly, the queue size can become larger than memory, resulting in cache misses, disk reads, and even slower performance. Backpressure can help by limiting the queue size, thereby maintaining a high throughput rate and good response times for jobs already in the queue. Once the queue fills up, clients get a server busy or HTTP 503 status code to try again later. Clients can retry the request at a later time, perhaps with exponential backoff strategy.

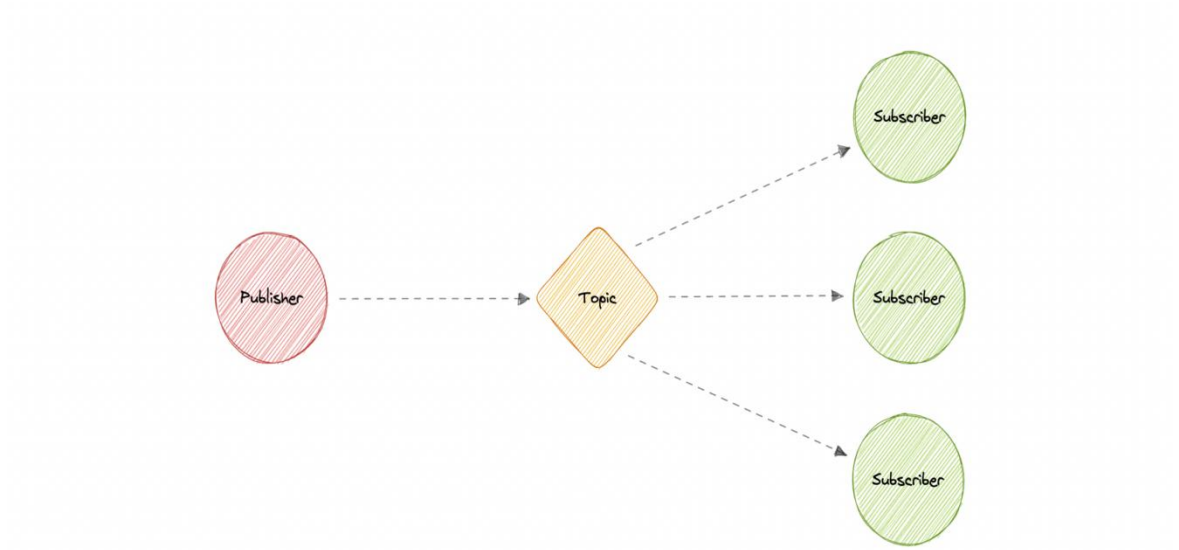
Examples

Following are some widely used message queues:

- Amazon SQS
- RabbitMQ
- ActiveMQ
- ZeroMQ

Publish-Subscribe

Similar to a message queue, publish-subscribe is also a form of service-to-service communication that facilitates asynchronous communication. In a pub/sub model, any message published to a topic is pushed immediately to all the subscribers of the topic.



The subscribers to the message topic often perform different functions, and can each do something different with the message in parallel. The publisher doesn't need to know who is using the information that it is broadcasting, and the subscribers don't need to know where the message comes from. This style of messaging is a bit different than message queues, where the component that sends the message often knows the destination it is sending to.

Working

Unlike message queues, which batch messages until they are retrieved, message topics transfer messages with little or no queuing and push them out immediately to all subscribers. Here's how it works:

- A message topic provides a lightweight mechanism to broadcast asynchronous event notifications and endpoints that allow software components to connect to the topic in order to send and receive those messages.
- To broadcast a message, a component called a publisher simply pushes a message to the topic.
- All components that subscribe to the topic (known as subscribers) will receive every message that was broadcasted.

Advantages

Let's discuss some advantages of using publish-subscribe:

- **Eliminate Polling**: Message topics allow instantaneous, push-based delivery, eliminating the need for message consumers to periodically check or "poll" for new information and updates. This promotes faster response time and reduces the delivery latency which can be particularly problematic in systems where delays cannot be tolerated.
- **Dynamic Targeting**: Pub/Sub makes the discovery of services easier, more natural, and less error-prone. Instead of maintaining a roster of peers where an application can send messages, a publisher will simply post messages to a topic. Then, any interested party will subscribe its endpoint to the topic, and start receiving these messages. Subscribers can change, upgrade, multiply or disappear and the system dynamically adjusts.
- **Decoupled and Independent Scaling**: Publishers and subscribers are decoupled and work independently from each other, which allows us to develop and scale them independently.
- **Simplify Communication**: The Publish-Subscribe model reduces complexity by removing all the point-to-point connections with a single connection to a message topic, which will manage subscriptions and decide what messages should be delivered to which endpoints.

Features

Now, let's discuss some desired features of publish-subscribe:

1. Push Delivery

Pub/Sub messaging instantly pushes asynchronous event notifications when messages are published to the message topic. Subscribers are notified when a message is available.

2. Multiple Delivery Protocols

In the Publish-Subscribe model, topics can typically connect to multiple types of endpoints, such as message queues, serverless functions, HTTP servers, etc.

3. Fanout

This scenario happens when a message is sent to a topic and then replicated and pushed to multiple endpoints. Fanout provides asynchronous event notifications which in turn allows for parallel processing.

4. Filtering

This feature empowers the subscriber to create a message filtering policy so that it will only get the notifications it is interested in, as opposed to receiving every single message posted to the topic.

5. Durability

Pub/Sub messaging services often provide very high durability, and at least once delivery, by storing copies of the same message on multiple servers.

6. Security

Message topics authenticate applications that try to publish content, this allows us to use encrypted endpoints and encrypt messages in transit over the network.

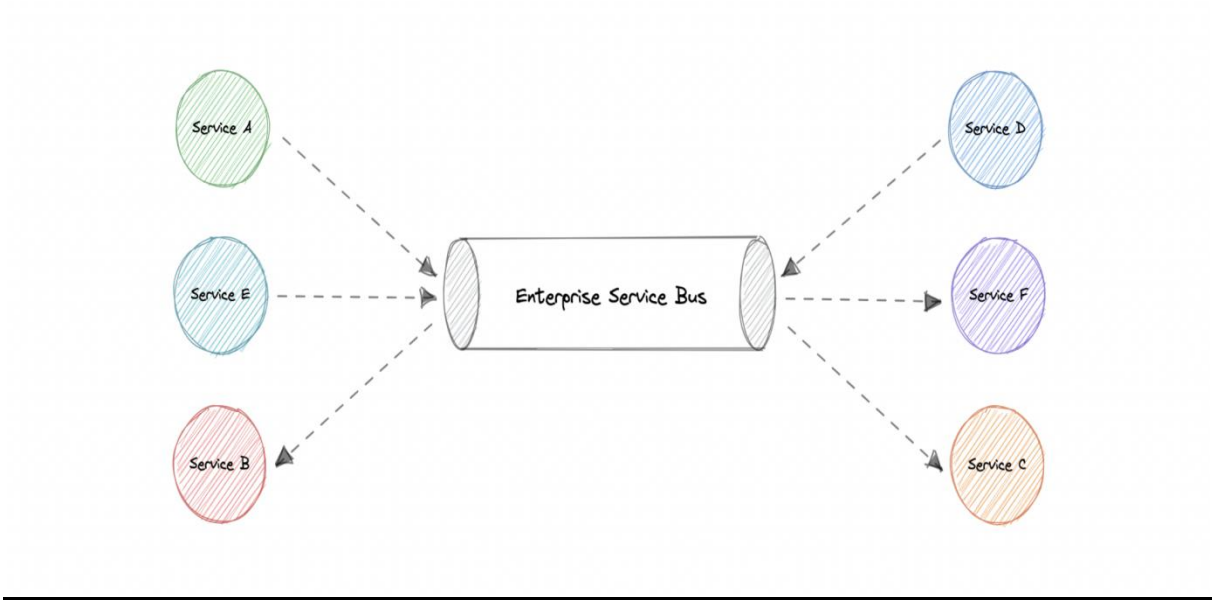
Examples

Here are some commonly used publish-subscribe technologies:

- Amazon SNS
- Google Pub/Sub

Enterprise Service Bus (ESB)

An Enterprise Service Bus (ESB) is an architectural pattern whereby a centralized software component performs integrations between applications. It performs transformations of data models, handles connectivity, performs message routing, converts communication protocols, and potentially manages the composition of multiple requests. The ESB can make these integrations and transformations available as a service interface for reuse by new applications.



Advantages

In theory, a centralized ESB offers the potential to standardize and dramatically simplify communication, messaging, and integration between services across the enterprise. Here are some advantages of using an ESB:

- **Improved developer productivity**: Enables developers to incorporate new technologies into one part of an application without touching the rest of the application.
- **Simpler, more cost-effective scalability**: Components can be scaled independently of others.

- **Greater resilience**: Failure of one component does not impact the others, and each microservice can adhere to its own availability requirements without risking the availability of other components in the system.

Disadvantages

While ESBs were deployed successfully in many organizations, in many other organizations the ESB came to be seen as a bottleneck. Here are some disadvantages of using an ESB:

- Making changes or enhancements to one integration could destabilize others who use that same integration.
- A single point of failure can bring down all communications.
- Updates to the ESB often impact existing integrations, so there is significant testing required to perform any update.
- ESB is centrally managed which makes cross-team collaboration challenging.
- High configuration and maintenance complexity.

Examples

Below are some widely used Enterprise Service Bus (ESB) technologies:

- Azure Service Bus
- IBM App Connect
- Apache Camel
- Fuse ESB

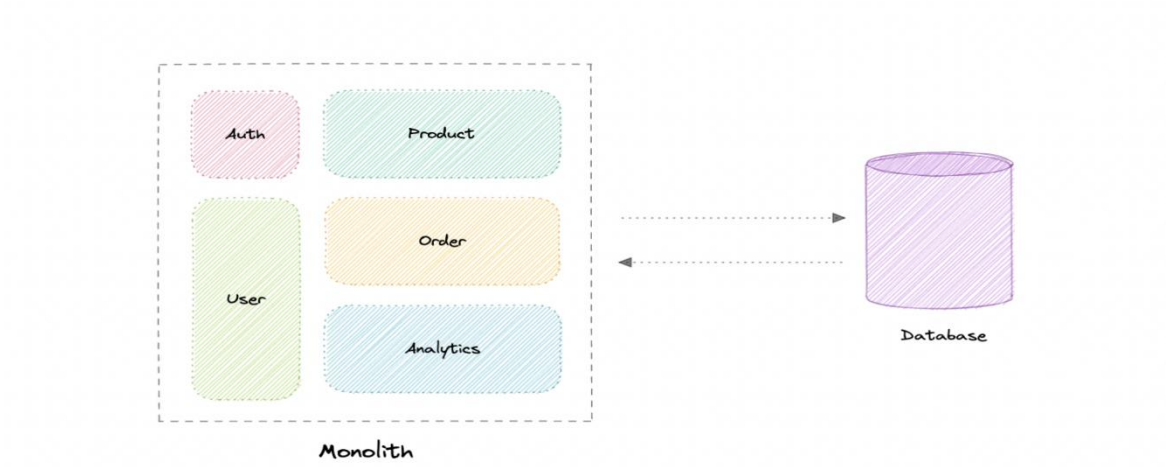
Sources and further reading

- It's all a numbers game
- Applying back pressure when overloaded
- Little's law
- What is the difference between a message queue and a task queue?

Monoliths and Microservices

Monoliths

A monolith is a self-contained and independent application. It is built as a single unit and is responsible for not just a particular task, but can perform every step needed to satisfy a business need.



Advantages

Following are some advantages of monoliths:

- Simple to develop or debug.
- Fast and reliable communication.
- Easy monitoring and testing.
- Supports ACID transactions.

Disadvantages

Some common disadvantages of monoliths are:

- Maintenance becomes hard as the codebase grows.
- Tightly coupled application, hard to extend.
- Requires commitment to a particular technology stack.
- On each update, the entire application is redeployed.
- Reduced reliability as a single bug can bring down the entire system.

- Difficult to scale or adopt new technologies.

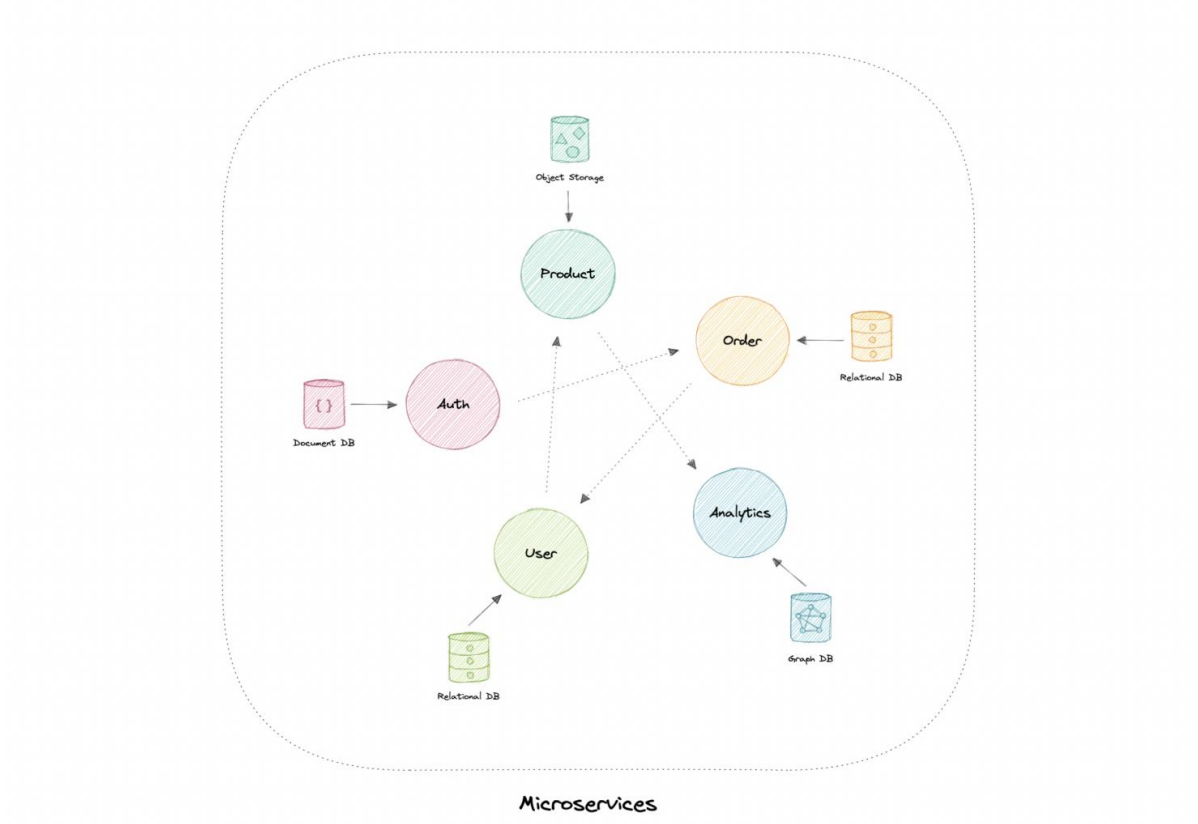
Modular Monoliths

A Modular Monolith is an approach where we build and deploy a single application (that's the Monolith part), but we build it in a way that breaks up the code into independent modules for each of the features needed in our application.

This approach reduces the dependencies of a module in such as way that we can enhance or change a module without affecting other modules. When done right, this can be really beneficial in the long term as it reduces the complexity that comes with maintaining a monolith as the system grows.

Microservices

Microservices architecture consists of a collection of small, autonomous services where each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division of business logic that provides an explicit boundary within which a domain model exists.



Each service has a separate codebase, which can be managed by a small development team. Services can be deployed independently and a team can update an existing service without rebuilding and redeploying the entire application.

Services are responsible for persisting their own data or external state (database per service). This differs from the traditional model, where a separate data layer handles data persistence.

Characteristics

The microservices architecture style has the following characteristics:

- **Loosely coupled**: Services should be loosely coupled so that they can be independently deployed and scaled. This will lead to the decentralization of development teams and thus, enabling them to develop and deploy faster with minimal constraints and operational dependencies.

- **Small but focused**: It's about scope and responsibilities and not size, a service should be focused on a specific problem. Basically, "It does one thing and does it well". Ideally, they can be independent of the underlying architecture.
- **Built for businesses**: The microservices architecture is usually organized around business capabilities and priorities.
- **Resilience & Fault tolerance**: Services should be designed in such a way that they still function in case of failure or errors. In environments with independently deployable services, failure tolerance is of the highest importance.
- **Highly maintainable**: Service should be easy to maintain and test because services that cannot be maintained will be rewritten.

Advantages

Here are some advantages of microservices architecture:

- Loosely coupled services.
- Services can be deployed independently.
- Highly agile for multiple development teams.
- Improves fault tolerance and data isolation.
- Better scalability as each service can be scaled independently.
- Eliminates any long-term commitment to a particular technology stack.

Disadvantages

Microservices architecture brings its own set of challenges:

- Complexity of a distributed system.
- Testing is more difficult.
- Expensive to maintain (individual servers, databases, etc.).
- Inter-service communication has its own challenges.
- Data integrity and consistency.
- Network congestion and latency.

Best practices

Let's discuss some microservices best practices:

- Model services around the business domain.
- Services should have loose coupling and high functional cohesion.
- Isolate failures and use resiliency strategies to prevent failures within a service from cascading.
- Services should only communicate through well-designed APIs. Avoid leaking implementation details.
- Data storage should be private to the service that owns the data
- Avoid coupling between services. Causes of coupling include shared database schemas and rigid communication protocols.
- Decentralize everything. Individual teams are responsible for designing and building services. Avoid sharing code or data schemas.
- Fail fast by using a circuit breaker to achieve fault tolerance.
- Ensure that the API changes are backward compatible.

Pitfalls

Below are some common pitfalls of microservices architecture:

- Service boundaries are not based on the business domain.
- Underestimating how hard is to build a distributed system.
- Shared database or common dependencies between services.
- Lack of Business Alignment.
- Lack of clear ownership.
- Lack of idempotency.
- Trying to do everything ACID instead of BASE.
- Lack of design for fault tolerance may result in cascading failures.

Beware of the distributed monolith

Distributed Monolith is a system that resembles the microservices architecture but is tightly coupled within itself like a monolithic application. Adopting microservices architecture comes with a lot of advantages. But while making one, there are good chances that we might end up with a distributed monolith.

Our microservices are just a distributed monolith if any of these apply to it:

- Requires low latency communication.
- Services don't scale easily.
- Dependency between services.
- Sharing the same resources such as databases.
- Tightly coupled systems.

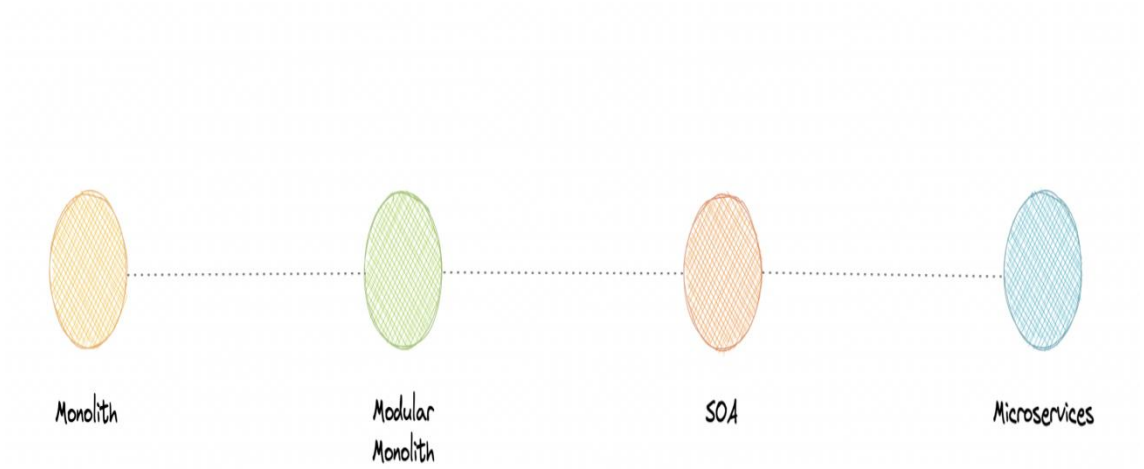
One of the primary reasons to build an application using microservices architecture is to have scalability. Therefore, microservices should have loosely coupled services which enable every service to be independent. The distributed monolith architecture takes this away and causes most components to depend on one another, increasing design complexity.

Microservices vs. Service-oriented architecture (SOA)

You might have seen Service-oriented architecture (SOA) mentioned around the internet, sometimes even interchangeably with microservices, but they are different from each other and the main distinction between the two approaches comes down to scope.

Service-oriented architecture (SOA) defines a way to make software components reusable via service interfaces. These interfaces utilize common communication standards and focus on maximizing application service reusability whereas microservices are built as a collection of various smallest independent service units focused on team autonomy and decoupling.

Why you don't need microservices



So, you might be wondering, monoliths seem like a bad idea to begin with, why would anyone use that?

Well, it depends. While each approach has its own advantages and disadvantages, it is advised to start with a monolith when building a new system. It is important to understand, that microservices are not a silver bullet, instead, they

solve an organizational problem. Microservices architecture is about your organizational priorities and team as much as it's about technology.

Before making the decision to move to microservices architecture, you need to ask yourself questions like:

- "Is the team too large to work effectively on a shared codebase?"
- "Are teams blocked on other teams?"
- "Does microservices deliver clear business value for us?"
- "Is my business mature enough to use microservices?"
- "Is our current architecture limiting us with communication overhead?"

If your application does not require to be broken down into microservices, you don't need this. There is no absolute necessity that all applications should be broken down into microservices.

We frequently draw inspiration from companies such as Netflix and their use of microservices, but we overlook the fact that we are not Netflix. They went through a lot of iterations and models before they had a market-ready solution, and this architecture became acceptable for them when they identified and solved the problem they were trying to tackle.

That's why it's essential to understand in-depth if your business actually needs microservices. What I'm trying to say is microservices are solutions to complex concerns and if your business doesn't have complex issues, you don't need them.

Event-Driven Architecture (EDA)

Event-Driven Architecture (EDA) is about using events as a way to communicate within a system. Generally, leveraging a message broker to publish and consume events asynchronously. The publisher is unaware of who is consuming an event and the consumers are unaware of each other. Event-Driven Architecture is simply a way of achieving loose coupling between services within a system.

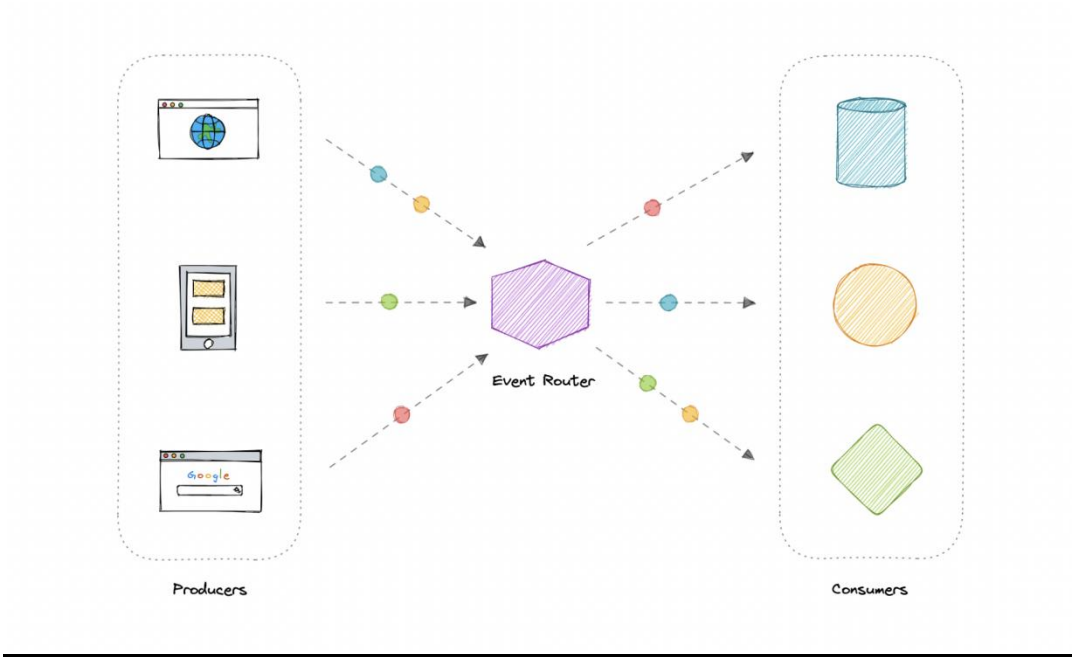
What is an event?

An event is a data point that represents state changes in a system. It doesn't specify what should happen and how the change should modify the system, it only notifies the system of a particular state change. When a user makes an action, they trigger an event.

Components

Event-driven architectures have three key components:

- **Event producers**: Publishes an event to the router.
- **Event routers**: Filters and pushes the events to consumers.
- **Event consumers**: Uses events to reflect changes in the system.



Note: Dots in the diagram represents different events in the system.

Patterns

There are several ways to implement the event-driven architecture, and which method we use depends on the use case but here are some common examples:

- Sagas
- Publish-Subscribe
- Event Sourcing
- Command and Query Responsibility Segregation (CQRS)

Note: Each of these methods is discussed separately.

Advantages

Let's discuss some advantages:

- Decoupled producers and consumers.
- Highly scalable and distributed.
- Easy to add new consumers.
- Improves agility.

Challenges

Here are some challenges of event-drive architecture:

- Guaranteed delivery.
- Error handling is difficult.
- Event-driven systems are complex in general.
- Exactly once, in-order processing of events.

Use cases

Below are some common use cases where event-driven architectures are beneficial:

- Metadata and metrics.
- Server and security logs.

- Integrating heterogeneous systems.
- Fanout and parallel processing.

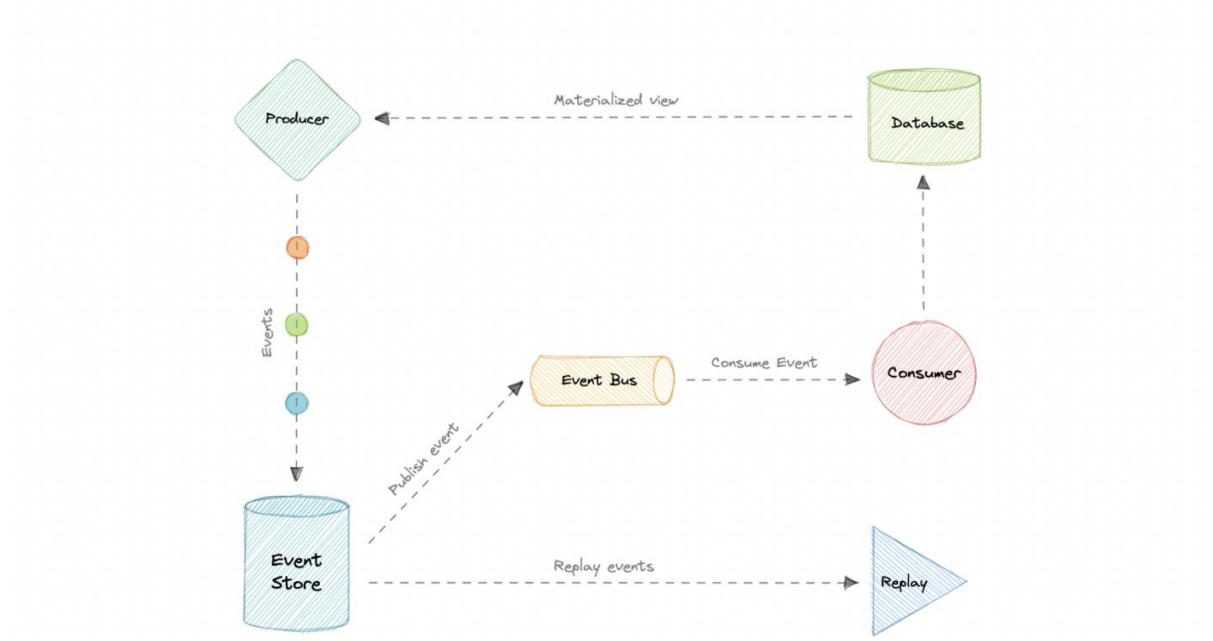
Examples

Here are some widely used technologies for implementing event-driven architectures:

- NATS
- Apache Kafka
- Amazon EventBridge
- Amazon SNS
- Google PubSub

Event Sourcing

Instead of storing just the current state of the data in a domain, use an append-only store to record the full series of actions taken on that data. The store acts as the system of record and can be used to materialize the domain objects.



This can simplify tasks in complex domains, by avoiding the need to synchronize the data model and the business domain, while improving performance, scalability, and responsiveness. It can also provide consistency for transactional data, and maintain full audit trails and history that can enable compensating actions.

Event sourcing vs Event-Driven Architecture (EDA)

Event sourcing is seemingly constantly being confused with Event-driven Architecture (EDA). Event-driven architecture is about using events to communicate between service boundaries. Generally, leveraging a message broker to publish and consume events asynchronously within other boundaries.

Whereas, event sourcing is about using events as a state, which is a different approach to storing data. Rather than storing the current state, we're instead going

to be storing events. Also, event sourcing is one of the several patterns to implement an event-driven architecture.

Advantages

Let's discuss some advantages of using event sourcing:

- Excellent for real-time data reporting.
- Great for fail-safety, data can be reconstituted from the event store.
- Extremely flexible, any type of message can be stored.
- Preferred way of achieving audit logs functionality for high compliance systems.

Disadvantages

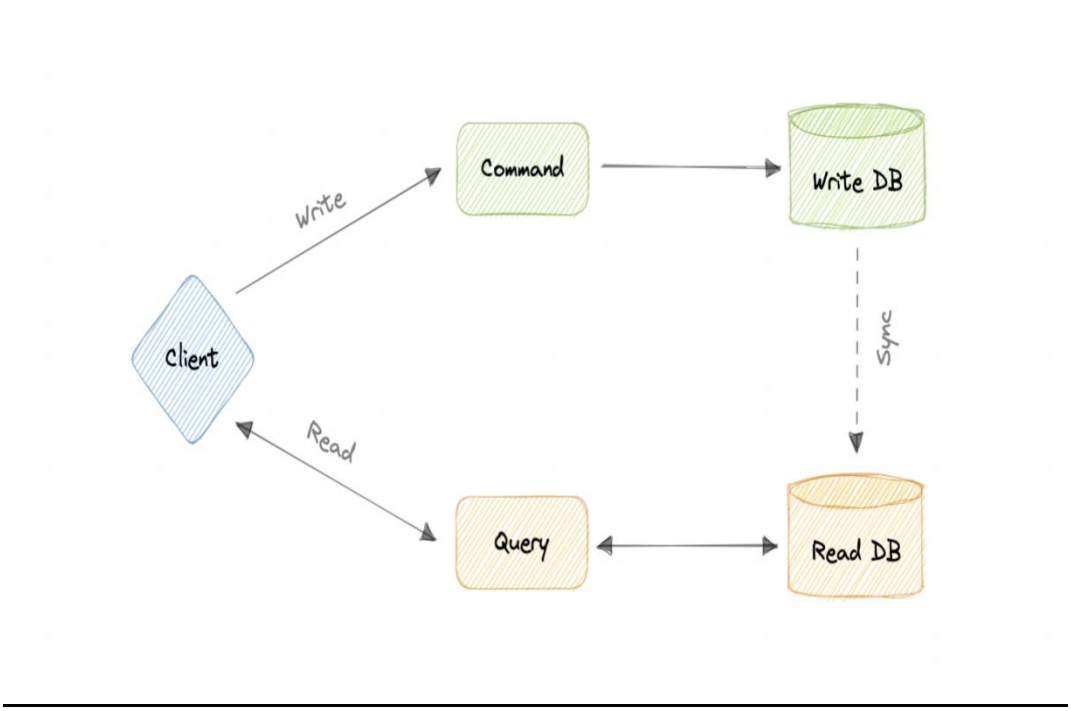
Following are the disadvantages of event sourcing:

- Requires an extremely efficient network infrastructure.
- Requires a reliable way to control message formats, such as a schema registry.
- Different events will contain different payloads.

Command and Query Responsibility Segregation (CQRS)

Command Query Responsibility Segregation (CQRS) is an architectural pattern that divides a system's actions into commands and queries. It was first described by Greg Young.

In CQRS, a command is an instruction, a directive to perform a specific task. It is an intention to change something and doesn't return a value, only an indication of success or failure. And, a query is a request for information that doesn't change the system's state or cause any side effects.



The core principle of CQRS is the separation of commands and queries. They perform fundamentally different roles within a system, and separating them means that each can be optimized as needed, which distributed systems can really benefit from.

CQRS with Event Sourcing

The CQRS pattern is often used along with the Event Sourcing pattern. CQRS-based systems use separate read and write data models, each tailored to relevant tasks and often located in physically separate stores.

When used with the Event Sourcing pattern, the store of events is the write model and is the official source of information. The read model of a CQRS-based system provides materialized views of the data, typically as highly denormalized views.

Advantages

Let's discuss some advantages of CQRS:

- Allows independent scaling of read and write workloads.
- Easier scaling, optimizations, and architectural changes.
- Closer to business logic with loose coupling.
- The application can avoid complex joins when querying.
- Clear boundaries between the system behavior.

Disadvantages

Below are some disadvantages of CQRS:

- More complex application design.
- Message failures or duplicate messages can occur.
- Dealing with eventual consistency is a challenge.
- Increased system maintenance efforts.

Use cases

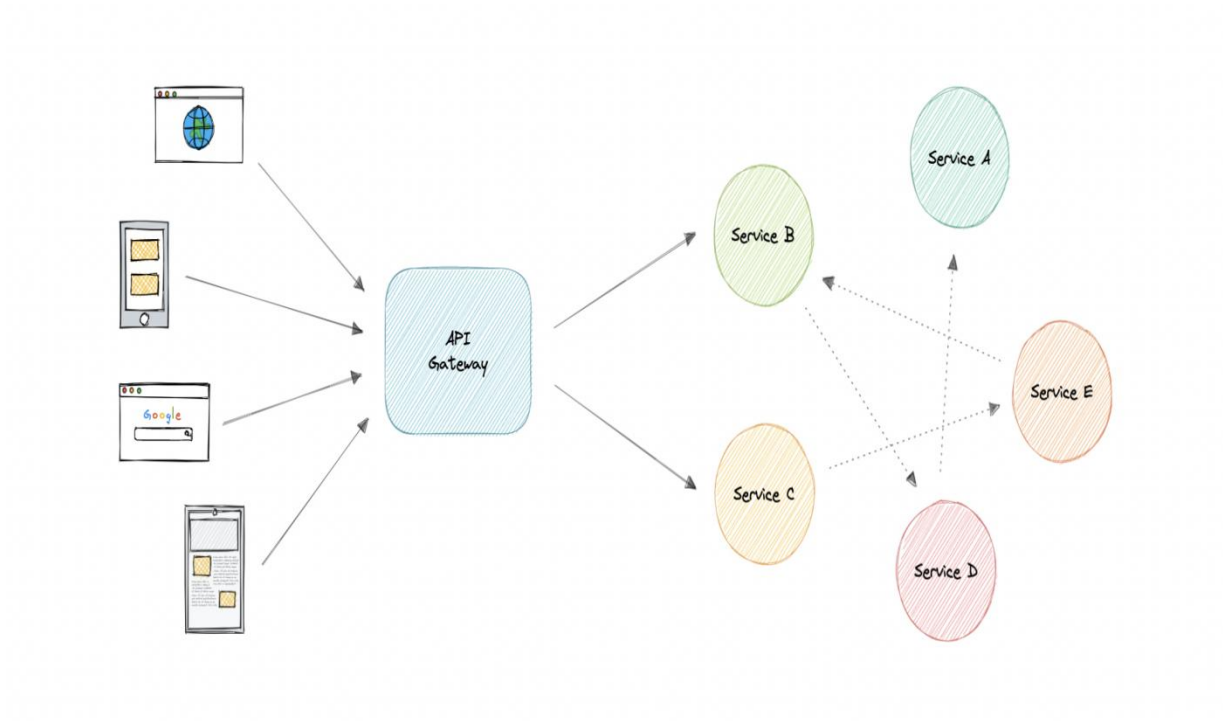
Here are some scenarios where CQRS will be helpful:

- The performance of data reads must be fine-tuned separately from the performance of data writes.
- The system is expected to evolve over time and might contain multiple versions of the model, or where business rules change regularly.
- Integration with other systems, especially in combination with event sourcing, where the temporal failure of one subsystem shouldn't affect the availability of the others.

- Better security to ensure that only the right domain entities are performing writes on the data.

API Gateway

The API Gateway is an API management tool that sits between a client and a collection of backend services. It is a single entry point into a system that encapsulates the internal system architecture and provides an API that is tailored to each client. It also has other responsibilities such as authentication, monitoring, load balancing, caching, throttling, logging, etc.



Why do we need an API Gateway?

The granularity of APIs provided by microservices is often different than what a client needs. Microservices typically provide fine-grained APIs, which means that clients need to interact with multiple services. Hence, an API gateway can provide a single entry point for all clients with some additional features and better management.

Features

Below are some desired features of an API Gateway:

- Authentication and Authorization

- Service discovery
- Reverse Proxy
- Caching
- Security
- Retry and Circuit breaking
- Load balancing
- Logging, Tracing
- API composition
- Rate limiting and throttling
- Versioning
- Routing
- IP whitelisting or blacklisting

Advantages

Let's look at some advantages of using an API Gateway:

- Encapsulates the internal structure of an API.
- Provides a centralized view of the API.
- Simplifies the client code.
- Monitoring, analytics, tracing, and other such features.

Disadvantages

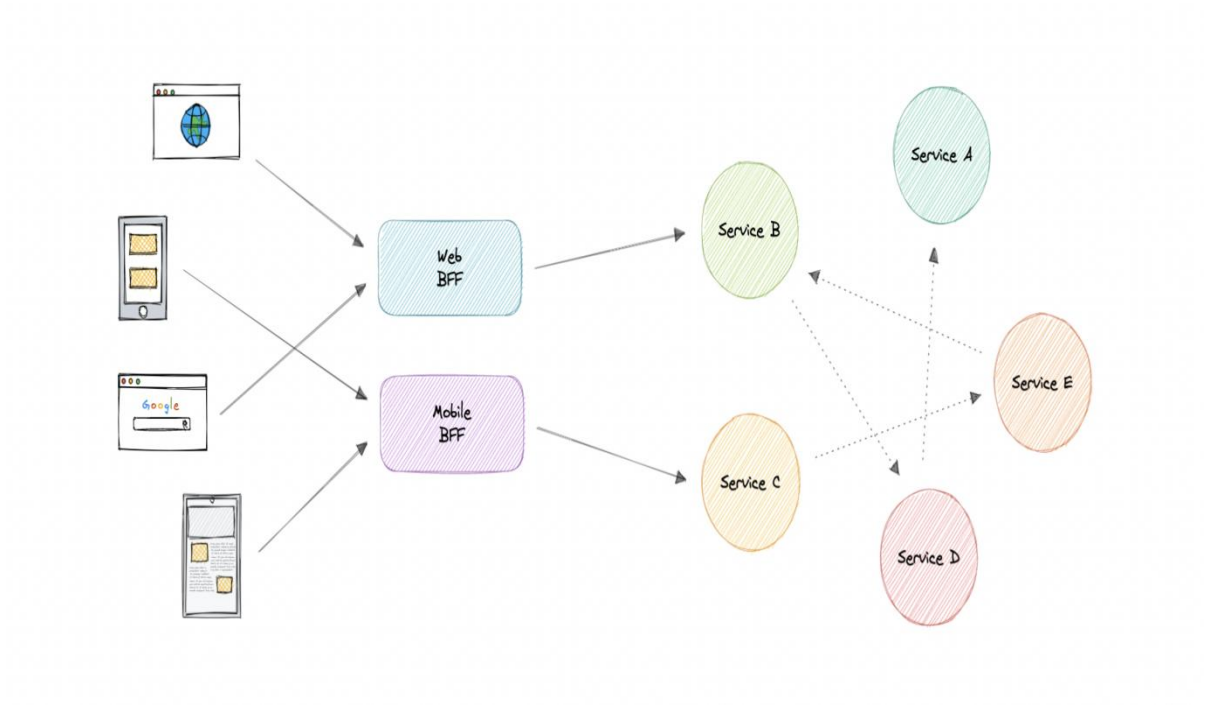
Here are some possible disadvantages of an API Gateway:

- Possible single point of failure.
- Might impact performance.
- Can become a bottleneck if not scaled properly.
- Configuration can be challenging.

Backend For Frontend (BFF) pattern

In the Backend For Frontend (BFF) pattern, we create separate backend services to be consumed by specific frontend applications or interfaces. This pattern is useful when we want to avoid customizing a single backend for multiple interfaces. This pattern was first described by Sam Newman.

Also, sometimes the output of data returned by the microservices to the front end is not in the exact format or filtered as needed by the front end. To solve this issue, the frontend should have some logic to reformat the data, and therefore, we can use BFF to shift some of this logic to the intermediate layer.



The primary function of the backend for the frontend pattern is to get the required data from the appropriate service, format the data, and sent it to the frontend. GraphQL performs really well as a backend for frontend (BFF).

When to use this pattern?

We should consider using a Backend For Frontend (BFF) pattern when:

- A shared or general purpose backend service must be maintained with significant development overhead.

- We want to optimize the backend for the requirements of a specific client.
- Customizations are made to a general-purpose backend to accommodate multiple interfaces.

Examples

Following are some widely used gateways technologies:

- Amazon API Gateway
- Apigee API Gateway
- Azure API Gateway
- Kong API Gateway

REST, GraphQL, gRPC

A good API design is always a crucial part of any system. But it is also important to pick the right API technology. So, in this tutorial, we will briefly discuss different API technologies such as REST, GraphQL, and gRPC.

What's an API?

Before we even get into API technologies, let's first understand what is an API.

API stands for Application Programming Interface. It is a set of definitions and protocols for building and integrating application software. It's sometimes referred to as a contract between an information provider and an information user establishing the content required from the producer and the content required by the consumer.

In other words, if you want to interact with a computer or system to retrieve information or perform a function, an API helps you communicate what you want to that system so it can understand and complete the request.

REST

A REST API (also known as RESTful API) is an application programming interface that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services. REST stands for Representational State Transfer and it was first introduced by Roy Fielding in the year 2000.

In REST API, the fundamental unit is a resource.

Concepts

Let's discuss some concepts of a RESTful API.

Constraints

In order for an API to be considered RESTful, it has to conform to these architectural constraints:

- **Uniform Interface**: There should be a uniform way of interacting with a given server.
- **Client-Server**: A client-server architecture managed through HTTP.
- **Stateless**: No client context shall be stored on the server between requests.
- **Cacheable**: Every response should include whether the response is cacheable or not and for how much duration responses can be cached at the client-side.
- **Layered system**: An application architecture needs to be composed of multiple layers.
- **Code on demand**: Return executable code to support a part of your application. (optional)

HTTP Verbs

HTTP defines a set of request methods to indicate the desired action to be performed for a given resource. Although they can also be nouns, these request methods are sometimes referred to as HTTP verbs. Each of them implements a different semantic, but some common features are shared by a group of them.

Below are some commonly used HTTP verbs:

- **GET**: Request a representation of the specified resource.
- **HEAD**: Response is identical to a GET request, but without the response body.
- **POST**: Submits an entity to the specified resource, often causing a change in state or side effects on the server.
- **PUT**: Replaces all current representations of the target resource with the request payload.
- **DELETE**: Deletes the specified resource.
- **PATCH**: Applies partial modifications to a resource.

HTTP response codes

HTTP response status codes indicate whether a specific HTTP request has been successfully completed.

There are five classes defined by the standard:

- 1xx - Informational responses.
- 2xx - Successful responses.
- 3xx - Redirection responses.
- 4xx - Client error responses.
- 5xx - Server error responses.

For example, HTTP 200 means that the request was successful.

Advantages

Let's discuss some advantages of REST API:

- Simple and easy to understand.
- Flexible and portable.
- Good caching support.
- Client and server are decoupled.

Disadvantages

Let's discuss some disadvantages of REST API:

- Over-fetching of data.
- Sometimes multiple round trips to the server are required.

Use cases

REST APIs are pretty much used universally and are the default standard for designing APIs. Overall REST APIs are quite flexible and can fit almost all scenarios.

Example

Here's an example usage of a REST API that operates on a **users** resource.

URI	HTTP verb	Description
/users	GET	Get all users
/users/{id}	GET	Get a user by id
/users	POST	Add a new user
/users/{id}	PATCH	Update a user by id
/users/{id}	DELETE	Delete a user by id

There is so much more to learn when it comes to REST APIs, I will highly recommend looking into Hypermedia as the Engine of Application State (HATEOAS).

GraphQL

GraphQL is a query language and server-side runtime for APIs that prioritizes giving clients exactly the data they request and no more. It was developed by Facebook and later open-sourced in 2015.

GraphQL is designed to make APIs fast, flexible, and developer-friendly. Additionally, GraphQL gives API maintainers the flexibility to add or deprecate fields without impacting existing queries. Developers can build APIs with whatever methods they prefer, and the GraphQL specification will ensure they function in predictable ways to clients.

In GraphQL, the fundamental unit is a query.

Concepts

Let's briefly discuss some key concepts in GraphQL:

Schema

A GraphQL schema describes the functionality clients can utilize once they connect to the GraphQL server.

Queries

A query is a request made by the client. It can consist of fields and arguments for the query. The operation type of a query can also be a mutation which provides a way to modify server-side data.

Resolvers

Resolver is a collection of functions that generate responses for a GraphQL query. In simple terms, a resolver acts as a GraphQL query handler.

Advantages

Let's discuss some advantages of GraphQL:

- Eliminates over-fetching of data.
- Strongly defined schema.

- Code generation support.
- Payload optimization.

Disadvantages

Let's discuss some disadvantages of GraphQL:

- Shifts complexity to server-side.
- Caching becomes hard.
- Versioning is ambiguous.
- N+1 problem.

Use cases

GraphQL proves to be essential in the following scenarios:

- Reducing app bandwidth usage as we can query multiple resources in a single query.
- Rapid prototyping for complex systems.
- When we are working with a graph-like data model.

Example

Here's a GraphQL schema that defines a User type and a Query type.

```
type Query {
        getUser: User
}
type User {
         id: ID
         name: String
         city: String
         state: String
}
```

Using the above schema, the client can request the required fields easily without having to fetch the entire resource or guess what the API might return.

```
{
	getUser {
		id
		name
		city
	}
}
```

This will give the following response to the client.

```
{
	"getUser": {
		"id": 123,
		"name": "Karan",
		"city": "San Francisco"
	}
}
```

Learn more about GraphQL at graphql.org.

gRPC

gRPC is a modern open-source high-performance Remote Procedure Call (RPC) framework that can run in any environment. It can efficiently connect services in and across data centers with pluggable support for load balancing, tracing, health checking, authentication and much more.

Concepts

Let's discuss some key concepts of gRPC.

Protocol buffers

Protocol buffers provide a language and platform-neutral extensible mechanism for serializing structured data in a forward and backward-compatible way. It's like JSON, except it's smaller and faster, and it generates native language bindings.

Service definition

Like many RPC systems, gRPC is based on the idea of defining a service and specifying the methods that can be called remotely with their parameters and return types. gRPC uses protocol buffers as the Interface Definition Language (IDL) for describing both the service interface and the structure of the payload messages.

Advantages

Let's discuss some advantages of gRPC:

- Lightweight and efficient.
- High performance.
- Built-in code generation support.
- Bi-directional streaming.

Disadvantages

Let's discuss some disadvantages of gRPC:

- Relatively new compared to REST and GraphQL.
- Limited browser support.
- Steeper learning curve.
- Not human readable.

Use cases

Below are some good use cases for gRPC:

- Real-time communication via bi-directional streaming.
- Efficient inter-service communication in microservices.

- Low latency and high throughput communication.
- Polyglot environments.

Example

Here's a basic example of a gRPC service defined in a *.proto file. Using this definition, we can easily code generate the HelloService service in the programming language of our choice.

```
service HelloService {
        rpc SayHello (HelloRequest) returns (HelloResponse);
}
message HelloRequest {
        string greeting = 1;
}
message HelloResponse {
        string reply = 1;
}
```

REST vs. GraphQL vs. gRPC

Now that we know how these API designing techniques work, let's compare them based on the following parameters:

- Will it cause tight coupling?
- How chatty (distinct API calls to get needed information) are the APIs?
- What's the performance like?
- How complex is it to integrate?
- How well does the caching work?
- Built-in tooling and code generation?
- What's API discoverability like?
- How easy is it to version APIs?

Type	Coupling	Chattiness	Performance	Complexity	Caching	Code gen	Discoverability	Versioning
REST	Low	High	Good	Medium	Great	Bad	Good	Easy
GraphQL	Medium	Low	Good	High	Custom	Good	Good	Custom
gRPC	High	Medium	Great	Low	Custom	Great	Bad	Hard

Which API technology is better?

Well, the answer is none of them. There is no silver bullet as each of these technologies has its own advantages and disadvantages. Users only care about using our APIs in a consistent way, so make sure to focus on your domain and requirements when designing your API.

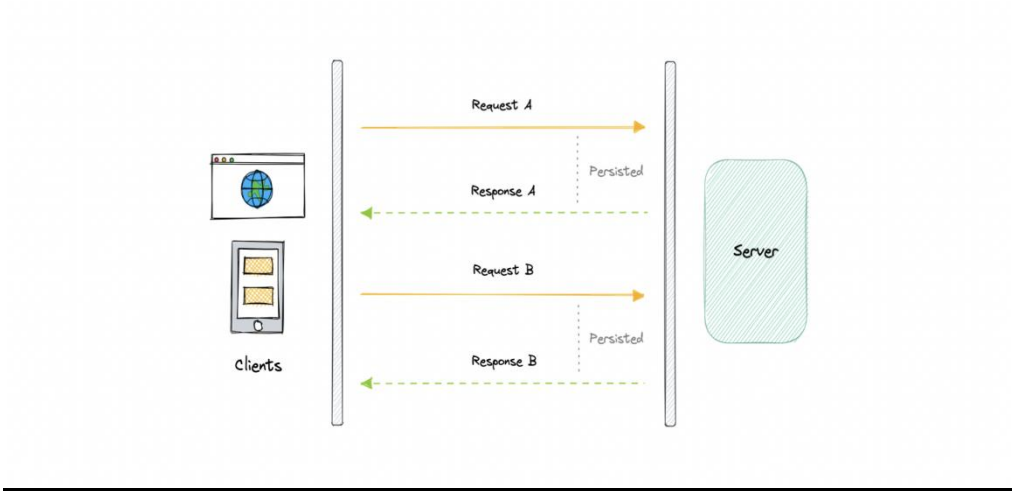
Long polling, Web Sockets, Server-Sent Events (SSE)

Web applications were initially developed around a client-server model, where the web client is always the initiator of transactions like requesting data from the server. Thus, there was no mechanism for the server to independently send, or push, data to the client without the client first making a request. Let's discuss some approaches to overcome this problem.

Long polling

HTTP Long polling is a technique used to push information to a client as soon as possible from the server. As a result, the server does not have to wait for the client to send a request.

In Long polling, the server does not close the connection once it receives a request from the client. Instead, the server responds only if any new message is available or a timeout threshold is reached.



Once the client receives a response, it immediately sends a new request to the server to have a new pending connection to send data to the client, and the operation is repeated. With this approach, the server emulates a real-time server push feature.

Working

Let's understand how long polling works:

1. The client makes an initial request and waits for a response.
2. The server receives the request and delays sending anything until an update is available.
3. Once an update is available, the response is sent to the client.
4. The client receives the response and makes a new request immediately or after some defined interval to establish a connection again.

Advantages

Here are some advantages of long polling:

- Easy to implement, good for small-scale projects.
- Nearly universally supported.

Disadvantages

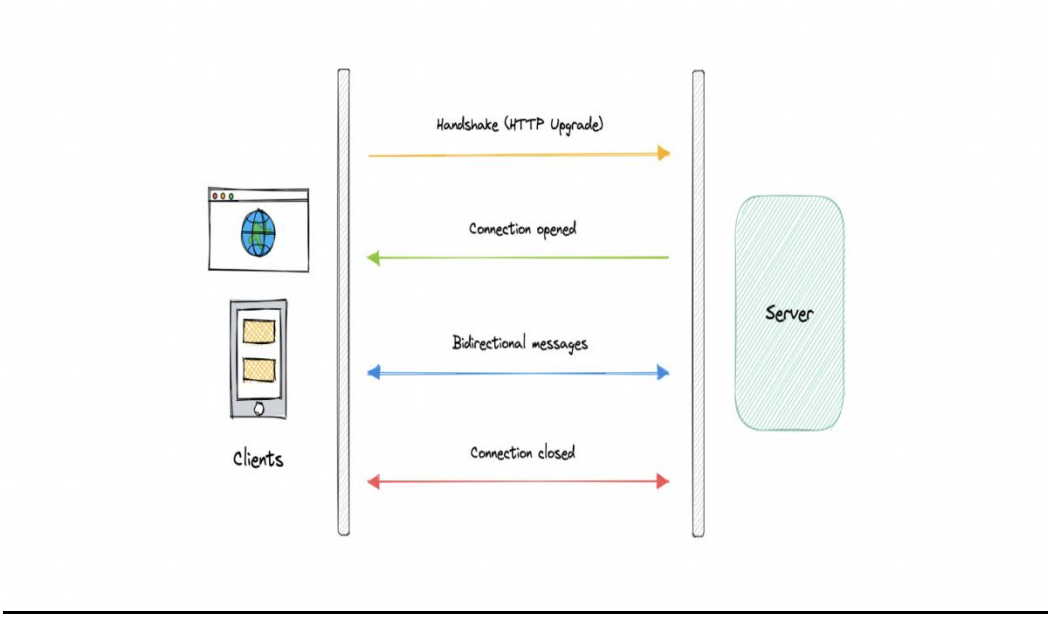
A major downside of long polling is that it is usually not scalable. Below are some of the other reasons:

- Creates a new connection each time, which can be intensive on the server.
- Reliable message ordering can be an issue for multiple requests.
- Increased latency as the server needs to wait for a new request.

Web Sockets

WebSocket provides full-duplex communication channels over a single TCP connection. It is a persistent connection between a client and a server that both parties can use to start sending data at any time.

The client establishes a WebSocket connection through a process known as the WebSocket handshake. If the process succeeds, then the server and client can exchange data in both directions at any time. The WebSocket protocol enables the communication between a client and a server with lower overheads, facilitating real-time data transfer from and to the server.



This is made possible by providing a standardized way for the server to send content to the client without being asked and allowing for messages to be passed back and forth while keeping the connection open.

Working

Let's understand how WebSockets work:

1. The client initiates a WebSocket handshake process by sending a request.
2. The request also contains an HTTP Upgrade header that allows the request to switch to the WebSocket protocol (ws://).
3. The server sends a response to the client, acknowledging the WebSocket handshake request.
4. A WebSocket connection will be opened once the client receives a successful handshake response.
5. Now the client and server can start sending data in both directions allowing real-time communication.
6. The connection is closed once the server or the client decides to close the connection.

Advantages

Below are some advantages of WebSockets:

- Full-duplex asynchronous messaging.
- Better origin-based security model.
- Lightweight for both client and server.

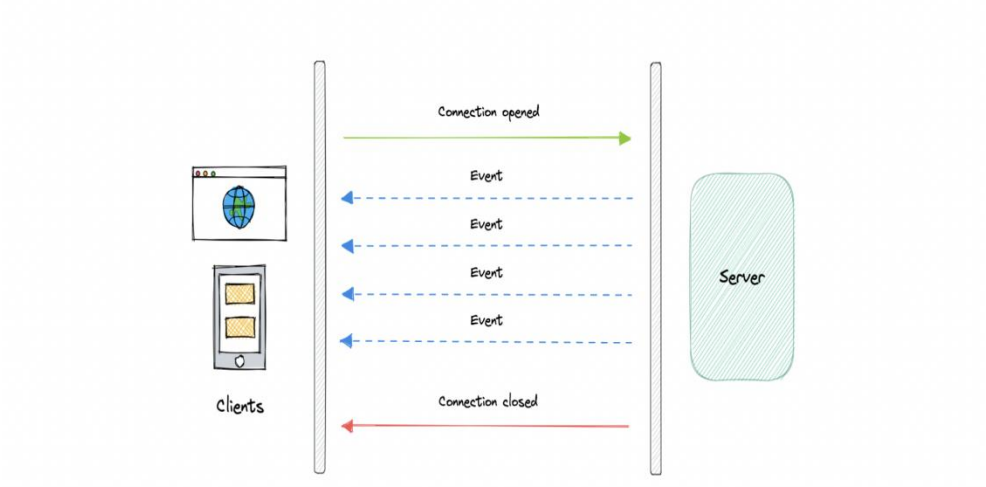
Disadvantages

Let's discuss some disadvantages of WebSockets:

- Terminated connections aren't automatically recovered.
- Older browsers don't support WebSockets (becoming less relevant).

Server-Sent Events (SSE)

Server-Sent Events (SSE) is a way of establishing long-term communication between client and server that enables the server to proactively push data to the client.



It is unidirectional, meaning once the client sends the request it can only receive the responses without the ability to send new requests over the same connection.

Working

Let's understand how server-sent events work:

1. The client makes a request to the server.
2. The connection between client and server is established and it remains open.
3. The server sends responses or events to the client when new data is available.

Advantages

- Simple to implement and use for both client and server.
- Supported by most browsers.
- No trouble with firewalls.

Disadvantages

- Unidirectional nature can be limiting.
- Limitation for the maximum number of open connections.
- Does not support binary data.

Merkel Tree

I recently hit upon the need to do check pointing in a data processing system that has the requirement that no data event can ever be lost and no events can be processed and streamed out of order. I wanted a way to auto-detect this in production in real time.

There are a couple of ways to do this, but since our data events already have a signature attached to them (a SHA1 hash), I decided that a useful way to do the checkpoint is basically keep a hash of hashes. One could do this with a hash list, where a chain of hashes for each data element is kept and when a checkpoint occurs the hash of all those hashes **in order** is taken.

A disadvantage of this model is if the downstream system detects a hash mismatch (either due to a lost message or messages that are out-of-order) it would then have to iterate the full list to detect where the problem is.

An elegant alternative is a hash tree, aka a **Merkle Tree** named after its inventor Ralph Merkle.

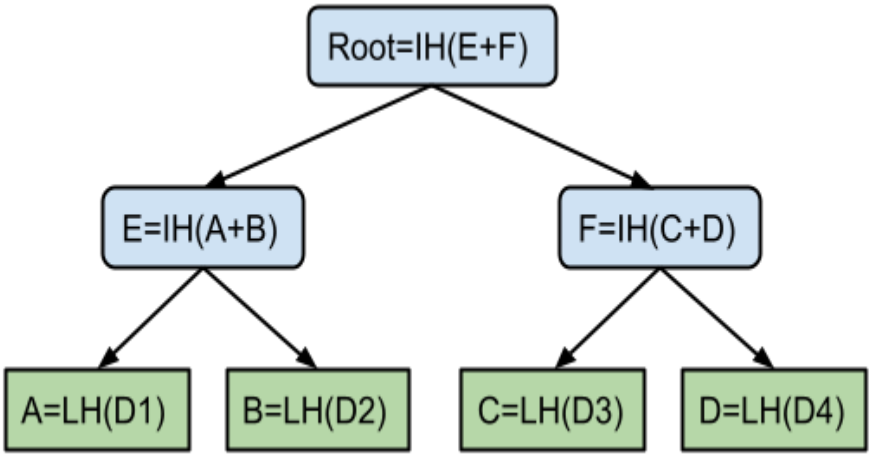
Merkle Trees

Merkle trees are typically implemented as binary trees where each non-leaf node is a hash of the two nodes below it. The leaves can either be the data itself or a hash/signature of the data.

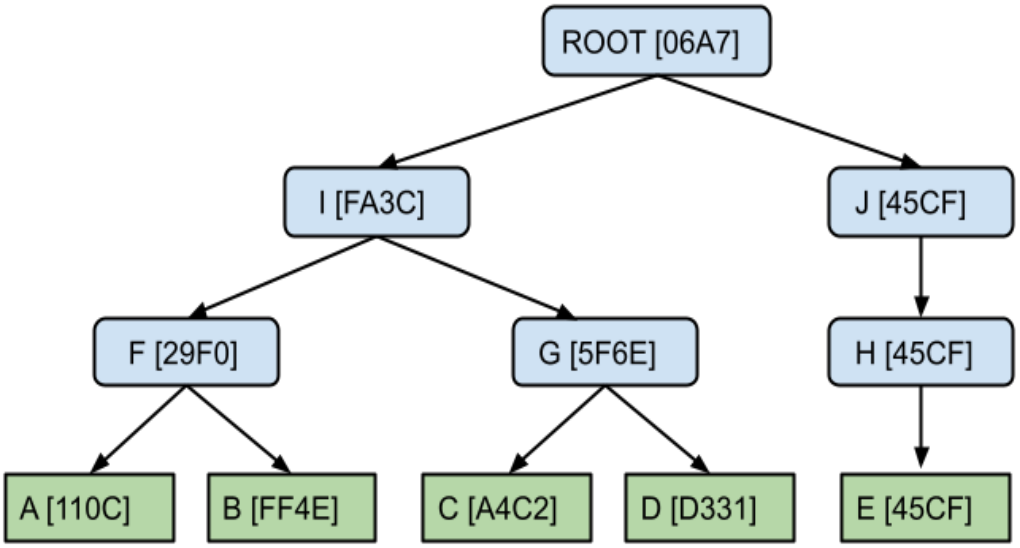
Thus, if any difference at the root hash is detected between systems, a binary search can be done through the tree to determine which particular subtree has the problem. Thus typically only log(N) nodes need to be inspected rather than all N nodes to find the problem area.

Merkle trees are particularly effective in distributed systems where two separate systems can compare the data on each node via a Merkle tree and quickly determine which data sets (subtrees) are lacking on one or the other system. Then only the subset of missing data needs to be sent. Cassandra, based on Amazon's Dynamo, for example, uses Merkle trees as an anti-entropy measure to detect inconsistencies between replicas.

The Tree Hash EXchange format (THEX) is used in some peer-to-peer systems for file integrity verification. In that system the internal (non-leaf) nodes are allowed to have a different hashing algorithm than the leaf nodes. In the diagram below IH=InternalHashFn and LH=LeafHashFn.



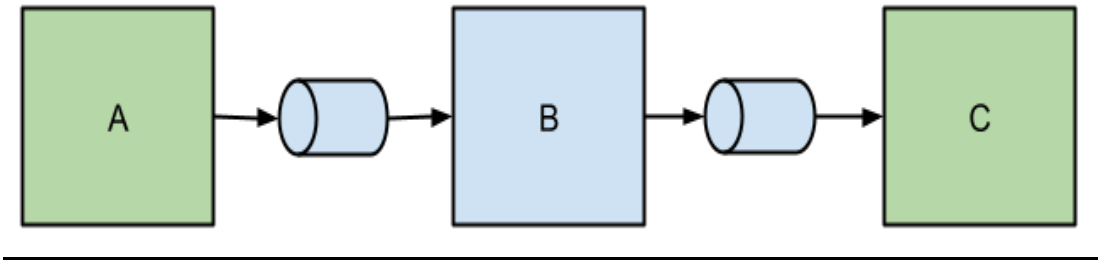
The THEX system also defines a serialization format and format for dealing with incomplete trees. The THEX system ensures that all leaves are at the same depth from the root node. To do that it "promotes" nodes. That is when a parent only has one child, it cannot does not take a hash of the child hash; instead it just "inherits" it. If that is confusing, think of the Merkle tree as being built from the bottom up: all the leaves are present and hashes of hashes are built until a single root is present.



Notation: The first token is a node label, followed by a conceptual value for the hash/signature of the node. Note that E, H and J nodes all have the same signature, since they only have one child node.

Merkle Tree as Checkpoint Data

Before I describe the implementation, it will help to see the use case I'm targeting.



The scenario above is a data processing pipeline where messages flow in one direction. All the messages that come out of A go into B and are processed and transformed to some new value-added structure and sent on to C. In between are queues to decouple the systems.

Throughput needs to be as high as possible and every message that comes out of A must be processed by B and sent to C in the same order. No data events can be lost or reordered. System A puts a signature (a SHA1 hash) into the metadata of the event and that metadata is present on the message event that C receives.

To ensure that all messages are received and in the correct order, a checkpoint is periodically created by A, summarizing all the messages sent since the last checkpoint. That checkpoint message is put onto the Queue between A and B; B passes it downstream without alteration so that C can read it. Between checkpoints, system C keeps a running list of all the events it has received so that it can compute the signatures necessary to validate what it has received against the checkpoint message that periodically comes in from A.

My Implementation of a Merkle Tree

The THEX Merkle Tree design was the inspiration for my implementation, but for my use case I made some simplifying assumptions. For one, I start with the leaves already having a signature. Since THEX is designed for file integrity comparisons, it assumes that you have segmented a file into fixed size chunks. That is not the use case I'm targeting.

The THEX algorithm "salts" the hash functions in order to ensure that there will be no collisions between the leaf hashes and the internal node hashes. It concatenates the byte 0x01 to the internal hash and the byte 0x00 to the leaf hash:

internal hash function = IH(X) = H(0x01, X)

leaf hash function = LH(X) = H(0x00, X)

It is useful to be able to distinguish leaf from internal nodes (especially when deserializing), so I morphed this idea into one where each Node has a type byte -- 0x01 identifies an internal node and 0x00 identifies a leaf node. This way I can leave the incoming leaf hashes intact for easier comparison by the downstream consumer.

So my MerkleTree.Node class is:

```
static class Node {
        public byte type;       // INTERNAL_SIG_TYPE or LEAF_SIG_TYPE
        public byte[] sig;       // signature of the node
        public Node left;
        public Node right;
}
```

Hash/Digest Algorithm

Since the leaf nodes are being passed in, my MerkleTree does not know (or need to know) what hashing algorithm was used on the leaves. Instead it only concerns itself with the internal leaf node digest algorithm.

The choice of hashing or digest algorithm is important, depending if you want to maximize performance or security. If one is using a Merkle tree to ensure integrity of data between peers that should not trust one another, then security is paramount and a cryptographically secure hash, such as SHA-256, Tiger, or SHA-3 should be used.

For my use case, I was not concerned with detecting malicious tampering. I only need to detect data loss or reordering, and have as little impact on overall throughput as possible. For that I can use a CRC rather than a full hashing algorithm.

Earlier I ran some benchmarks comparing the speed of Java implementations of SHA-1, Guava's Murmur hash, CRC32 and Adler32. Adler32 (java.util.zip.Adler32) was the fastest of the bunch. The typical use case for the Adler CRC is to detect data transmission errors. It trades off reliability for speed, so it is the weakest choice, but I deemed it sufficient to detect the sort of error I was concerned with.

So in my implementation the Adler32 checksum is hard-coded into the codebase. But if you want to change that we can either make the internal digest algorithm injectable or configurable or you can just copy the code and change it to use the algorithm you want.

The rest of the code is written to be agnostic of the hashing algorithm - all it deals with are the bytes of the signature.

Serialization / Deserialization

My implementation has efficient binary serialization built into the MerkleTree and an accompanying MerkleDeserializer class that handles the deserialization.

I chose not to use the Java Serialization framework. Instead the serialize method just returns an array of bytes and deserialize accepts that byte array.

The serialization format is:

```
(magicheader:int)(numnodes:int)
[(nodetype:byte)(siglength:int)(signature:[]byte)]
```

Where (foo:type) indicates the name (foo) and the type/size of the serialized element. I use a magic header of 0xcdaace99 to allow the deserializer to be certain it has received a valid byte array.

The next number indicates the number of nodes in the tree. Then follows an "array" of numnodes size where the elements are the node type (0x01 for internal, 0x00 for leaf), the length of the signature and then the signature as an array of bytes siglength long.

By including the siglength field, I can allow leaf nodes signatures to be "promoted" to the parent internal node when there is an odd number of leaf nodes. This allows the internal nodes to use signatures of different lengths.

Usage

For the use case described above, you can imagine that system A does the following:

```
List<String> eventSigs = new ArrayList<>();
while (true) {
        Event event = receiveEvent();
        String hash = computeHash(event);
```

```
        // ... process and transmit the message to the downstream Queue
        sendToDownstreamQueue(hash, event);

        eventSigs.add(has);

        if (isTimeForCheckpoint()) {
                MerkleTree mtree = new MerkleTree(eventSigs);
                eventSigs.clear();
                byte[] serializedTree = mtree.serialize();
                sendToDownstreamQueue(serializedTree);
        }
}
```

And system C would then do something like:

```
List<String> eventSigs = new ArrayList<>();
while (true) {
        Event event = receiveEvent();

        if (isCheckpointMessage(event)) {
        MerkleTree mytree = new MerkleTree(eventSigs);
        eventSigs.clear();

        byte[] treeBytes = event.getDataAsBytes();
        MerkleTree expectedTree = MerkleDeserializer.deserialize(treeBytes);
        byte[] myRootSig = mytree.getRoot().sig;
        byte[] expectedRootSig = expectedTree.getRoot().sig;
        if (!signaturesAreEqual(myRootSig, expectedRootSig)) {
                evaluateTreeDifferences(mytree, expectedTree);
                // ... send alert
```

```
            }
            else {
                        String hash = event.getOriginalSignature();
                        eventSigs.add(hash);
                        // .. do something with event
            }
}
```

go-gossip

This is Go implementation of the Gossip protocol.

Gossip is a communication protocol that delivers messages in a distributed system.

The point is that each node transmits data while periodically exchanging metadata based on TCP/UDP without a broadcast master.

In general, each node periodically performs health checks of other nodes and communicates with them, but this library relies on an externally imported discovery layer.

The gossip protocol is divided into two main categories: Push and Pull. If implemented as a push, it becomes inefficient if a large number of peers are already infected. If implemented as Pull, it will propagate efficiently, but there is a point to be concerned about because the message that needs to be propagated to the peer needs to be managed.

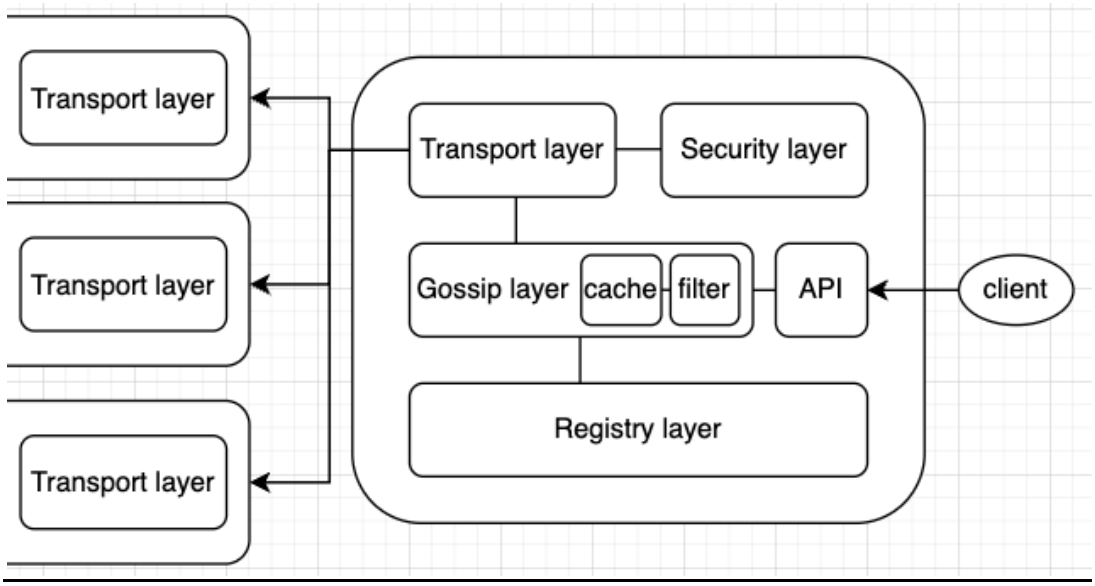
This project implements the Pull-based Gossip protocol. That's why we need to implement a way to send a new message when another node requests it. In this library, it consists of two parts, 'filter' that checks whether a message is received and 'cache' that stores the message for propagation.

Take a look at the list of supported features below.

- Message propagation
- Secure transport

It's focus on gossip through relying on discovery layer from outside.

Layer



Registry layer

Registry layer serves as the managed peer table. That could be static peer table, also could dynamic peer table (like DHT).

The required (MUST) method is Gossipiers. Gossipiers is used to select random peers to send gossip messages to.

```
type Registry interface {

	Gossipiers() []string

}
```

(It means DHT or importers covers registration to Gossip protocol)

Gossipiers returns array of raw addresses. The raw addresses will validate when gossip layer. Even if rely on validation of externally imported methods, We need to double-check internally here(trust will make unexpected panic).

A consideration is whether Gossipiers return value actually needs a peer ID. In generally, there is need each peer's meta datas(about memberlist), the unique id is required. However, since this library does not suppo

rt health checks, the peer id is not required from a metadata required point of view.In addition, if you receive and use a unique ID from outside, the dependency relationship becomes severe, so I think it is correct not to have an peer id.

When making a request, such as checking gossip node stat or something, we decide to use the raw address.

Gossip layer

Gossip layer serves core features that propagating gossip messages and relay data to application when needed.

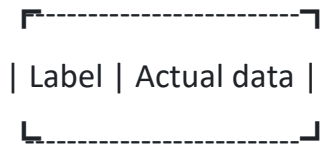
For serve that, it's detect packet and handles them correctly to the packet types.

There is three tasks what this layer have to do.

1. The node should be able to be a source of gossip. Provide surface interface for application programs to push gossip messages.
2. Handles them correctly to the packet types.
3. Detects is the gossip message already exist in memory and relay the gossip messages to the application if necessary.

Take a look the packet specification below.

Packet



Label

Packet type (1 byte)

1: PullRequest

2: PullResponse

Packet handle

- **PullRequest** - It replies a packet to the requester. However, packets that have already been taken by the requester are excluded.
- **PullResponse** - Stores the received message in memory. Messages that have already been received will be ignored.

Filter and Propagator

Filter is used to distinguish between 'already received messages' and 'newly received messages'.

To determine this 'accurately' in an asynchronous network environment, all history must be stored. Therefore, if you are sensitive to receiving duplicate data,

store the history permanently (currently implemented as LevelDB). If it's not very sensitive, store it in a large enough memory cache.

Propagator classifies newly received messages using filters and manages a cache to propagate them to other nodes. Actually, even if all messages are propagated, nodes use filters to store only those messages that have not been received. However, we exclude and propagate messages that are determined to have propagated 'enough' to reduce network congestion.

Transport/Security layer

Transport layer supports peer-to-peer UDP communication. Security layer resolve the secure between peer to peer trnasmission. From the point of view of packet fragmentation, to use UDP, packets must be divided and transmitted at the application level or TCP must be used, but this library does not care about packet fragmentation. The maximum packet size is set by subtracting the size of the IP header (20 bytes) and UDP header (8 bytes) from the 2^18 bytes written to the UDP header. So, max packet size is 65507 byte.

It should be possible to add multiple encryption algorithms. I'm just considering a method of encrypting and decrypting using a passphrase(It is also okay to encrypt in the application and then propagate it. In this case, you should set NO-SECURE in config).