When to use?

How to break monolith into microservices?

Data redundancy

Replication

Scaling (horizontal, vertical)

Load balancing

Microservices vs SOA

Saga pattern

Trace Id

# CAP theorem

The CAP theorem, also known as Brewer's theorem, is a fundamental concept in distributed systems that was formulated by computer scientist Eric Brewer in 2000. The theorem states that in a distributed system, it is impossible to simultaneously achieve all three of the following guarantees:

1. **Consistency (C)**: Every read operation on the system returns the most recent write, or an error. In other words, all nodes in the distributed system have a consistent view of the data at all times.
2. **Availability (A)**: Every request (read or write) to the system receives a response, without guaranteeing that it contains the most up-to-date data. The system is always responsive and accessible to clients.
3. **Partition Tolerance (P)**: The system continues to operate correctly in the presence of network partitions or communication failures between nodes. Network partitions can occur due to network issues, hardware failures, or other reasons, leading to a situation where some nodes in the system can't communicate with others.

The CAP theorem essentially asserts that in a distributed system, you can pick any two of these guarantees, but you can't have all three simultaneously. This theorem has significant implications for the design and operation of distributed systems.

Here are the possible trade-offs and implications:

1. **CA Systems (Consistency and Availability)**: In scenarios where data consistency and immediate availability are critical, you can sacrifice partition tolerance. These systems prioritize ensuring that all nodes have the most recent data and are available to serve requests. However, they are vulnerable to network partitions because they might become unavailable when network issues arise.
2. **CP Systems (Consistency and Partition Tolerance)**: In scenarios where data consistency and fault tolerance are essential, you can sacrifice immediate availability. These systems prioritize maintaining data consistency even in the presence of network partitions, which may result in temporarily unresponsive nodes during partition events.
3. **AP Systems (Availability and Partition Tolerance)**: In scenarios where high availability and fault tolerance are paramount, you can sacrifice strong consistency. These systems prioritize always being available to serve requests, even if it means that different nodes might have slightly inconsistent views of the data during network partitions.

Balancing consistency, availability, and partition tolerance is a complex task that depends on the specific requirements of your distributed system and its use cases. Some strategies to achieve a balance include:

* **Tuning Configuration Parameters**: Depending on the distributed database or system you're using, you can adjust configuration parameters to favor one aspect of the CAP triangle over the others. For example, you can configure replication factors or consistency levels in databases like Apache Cassandra.
* **Architectural Choices**: You can design your system's architecture to mitigate the impact of network partitions, such as by using quorum-based approaches, leader-follower models, or consensus algorithms like Raft or Paxos.
* **Use Case-Specific Strategies**: Different use cases within the same distributed system might have varying requirements. You can apply different CAP trade-offs to different parts of your system based on their specific needs.
* **Hybrid Approaches**: In some cases, you might employ hybrid strategies that adapt dynamically to the network conditions. For example, during periods of network stability, you can prioritize consistency, and during network partitions, you can relax consistency to ensure availability.

Ultimately, achieving the right balance among consistency, availability, and partition tolerance requires careful consideration of your system's requirements and trade-offs. There's no one-size-fits-all solution, and the appropriate approach will depend on the specific goals and constraints of your distributed system.

# Service discovery

Service discovery in the context of microservices refers to the dynamic process by which individual microservices can find and communicate with each other within a distributed system. This is a crucial component of building and operating microservices-based applications, and it serves several important purposes:

1. **Dynamic Nature of Microservices:** Microservices are designed to be independently deployable and scalable. They may be distributed across multiple servers or containers. As a result, their locations (IP addresses, ports, etc.) can change frequently due to scaling, failures, or updates. Service discovery helps microservices adapt to this dynamic environment.
2. **Load Balancing:** Service discovery can be used to implement load balancing strategies. When multiple instances of a service are available, a service discovery mechanism can evenly distribute incoming requests among them, improving performance and fault tolerance.
3. **Failover and Redundancy:** In case a microservice instance fails, service discovery can automatically redirect traffic to healthy instances, minimizing downtime and service disruption. This is critical for maintaining high availability.
4. **Service Composition:** Microservices often need to interact with other services to perform complex tasks. Service discovery helps them locate and establish connections with the required services, enabling seamless communication between microservices.
5. **Simplified Configuration:** Without service discovery, microservices would need to maintain a static list of IP addresses and ports for all the services they depend on. Service discovery simplifies this by allowing microservices to query a central registry or service discovery server to obtain the necessary information dynamically.
6. **Scaling:** As the number of microservices instances grows or shrinks based on demand, service discovery helps each microservice instance discover and communicate with its peers without manual intervention. This supports efficient auto-scaling.

Service discovery can be implemented using various tools and approaches. Some popular options include:

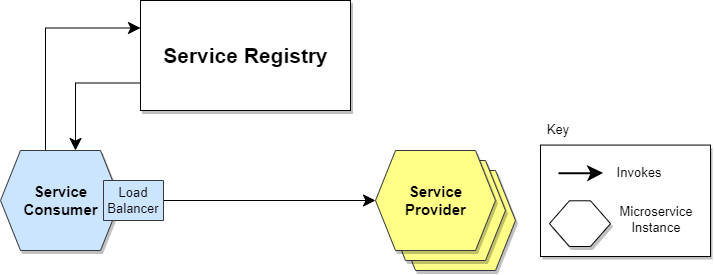
* **Service Registries:** Maintaining a central registry that contains information about all available microservices, including their locations and health status. Tools like Consul, ZooKeeper, and Eureka are examples of service registry systems.
* **DNS-Based Discovery:** Using DNS to resolve service names to their corresponding IP addresses and ports. This approach can be simpler to set up but may have limitations in terms of load balancing and failover.
* **API Gateway:** Implementing service discovery at the API gateway level, where incoming requests are routed to the appropriate microservices based on a configuration or routing rules.

In summary, service discovery is essential in microservices architecture because it enables the dynamic, efficient, and reliable communication between microservices in a constantly changing and distributed environment, ensuring that the system can scale, handle failures, and deliver a high-quality user experience.

### 5.1. Client-Side Service Discovery

When using Client-Side Discovery, **the Service Consumer is responsible for determining the network locations of available service instances and load balancing requests between them.** The client queries the Service Register. Then the client uses a load-balancing algorithm to choose one of the available service instances and performs a request.

The following diagram shows the pattern just described:

[](https://www.baeldung.com/wp-content/uploads/sites/4/2022/01/Service-Discovery-Client-Side.png)

Giving responsibility for client-side load balancing is both a burden and an advantage.**It’s an advantage because it saves an extra hop that we would’ve had with a dedicated load balancer. It’s a disadvantage because the Service Consumer must implement the load balancing logic.**

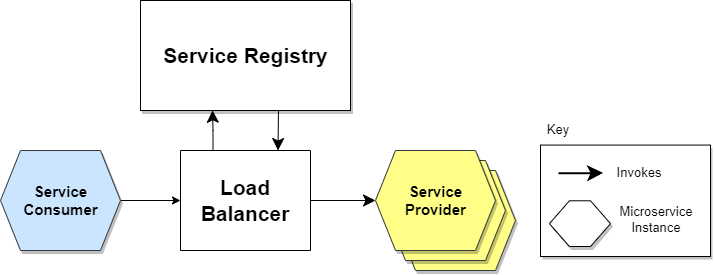
We can also point out that the Service Consumer and the Service Registry are quite coupled. This means that Client-Side Discovery logic must be implemented for each programming language and framework used by the Service Consumers.

Now that we’ve clarified Client-Side Discovery, let’s take a look at Server-Side Discovery.

### 5.2. Server-Side Service Discovery

**The alternate approach to Service Discovery is the Server-Side Discovery model, which uses an intermediary that acts as a**[**Load Balancer**](https://www.baeldung.com/zuul-load-balancing)**.** The client makes a request to a service via a load balancer that acts as an orchestrator. The load balancer queries the Service Registry and routes each request to an available service instance.

The following diagram shows how communication takes place:

[](https://www.baeldung.com/wp-content/uploads/sites/4/2022/01/Service-Discovery-Server-Side.png)

In this approach, a dedicated actor, the Load Balancer, does the job of load balancing. This is the main advantage of this approach. Indeed, **creating this level of abstraction makes the Service Consumer lighter, as it doesn’t have to deal with the lookup procedure.** As a matter of fact, there’s no need to implement the discovery logic separately for each language and framework that the Service Consumer uses.

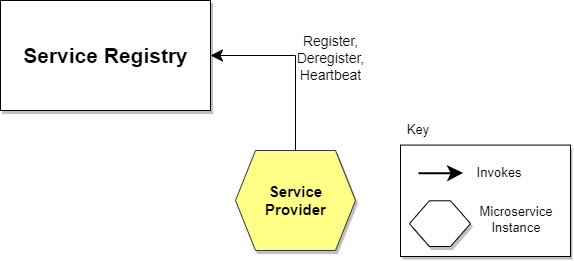
**On the other hand, we must set up and manage the Load Balancer, unless it’s already provided in the deployment environment.**

Now that we’ve delved into the different approaches to the discovery mechanisms, let’s move on to registration mechanisms.

## 7. Service Registration Options

### 7.1. Self-Registration

**When using the self-registration model, a service instance is responsible for registering and de-registering itself in the Service Registry.** In addition, if necessary, a service instance sends [heartbeat](https://martinfowler.com/articles/patterns-of-distributed-systems/heartbeat.html) requests to keep its registration alive. The following diagram shows the structure of this pattern:

[](https://www.baeldung.com/wp-content/uploads/sites/4/2022/01/Service-Discovery-Self-Registration.png)

The self-registration model has several pros and cons. One advantage is that it’s relatively simple and doesn’t require other system components as intermediaries. However, a significant disadvantage is that it couples service instances to the Service Registry, which means we must implement the registration code in each language and framework used.

An alternate approach, which decouples services from the Service Registry, is the third-party registration scheme.

# Role of Docker and Kubernetes

**1. Containers (e.g., Docker):**

* **Isolation:** Containers provide a lightweight form of virtualization that isolates applications and their dependencies from the underlying infrastructure. Each microservice can run in its own container, ensuring that it has everything it needs to function correctly without interfering with other services.
* **Consistency:** Containers package the application code, runtime, libraries, and configuration into a single, consistent unit. This eliminates the "it works on my machine" problem and ensures that the microservice behaves the same way across different environments, from development to production.
* **Portability:** Containers are highly portable. You can run the same containerized microservice on different cloud providers or on-premises infrastructure, which simplifies deployment and reduces vendor lock-in.
* **Scalability:** Containers are well-suited for horizontal scaling. You can easily replicate containers to handle increased loads, and container platforms provide tools for managing container lifecycles.

**2. Container Orchestration (e.g., Kubernetes):**

* **Cluster Management:** Kubernetes and similar container orchestration platforms manage clusters of machines where containers run. They automate tasks like deploying, scaling, updating, and monitoring containers across the cluster.
* **Service Discovery:** Container orchestration tools often include service discovery mechanisms that help microservices discover and communicate with each other. Kubernetes, for instance, offers DNS-based service discovery and load balancing.
* **Scaling:** Container orchestration platforms enable auto-scaling, which means you can automatically adjust the number of running microservice instances based on resource utilization or other metrics. This ensures that your application can handle varying workloads efficiently.
* **Rolling Updates:** Orchestrators facilitate rolling updates, allowing you to update microservices without causing downtime. New containers can be gradually rolled out while old ones are phased out, ensuring a smooth transition.
* **Health Checks and Self-healing:** Container orchestration platforms continuously monitor the health of microservice instances. If an instance becomes unhealthy (e.g., due to crashes or unresponsiveness), the orchestrator can automatically replace it with a healthy one.
* **Resource Management:** Orchestrators allocate resources (CPU, memory, etc.) to containers based on defined policies. This ensures that microservices have the necessary resources to operate efficiently.
* **Configuration Management:** Many orchestration tools support configuration management, allowing you to manage environment-specific configurations for microservices. This is particularly important in microservices deployments where each service may have unique requirements.
* **Rollback:** In case of issues during updates or deployments, container orchestrators offer rollback mechanisms to revert to a previous, stable state of the application.

Overall, containers and container orchestration provide a robust and scalable infrastructure for microservices, enabling developers to focus on writing code while automating the management of containerized microservices, making it easier to build, deploy, and maintain complex distributed systems. Kubernetes is one of the most popular container orchestration platforms, but there are others like Docker Swarm, Amazon ECS, and Apache Mesos, each with its own features and capabilities.

# What strategies can be employed for ensuring data consistency and synchronization across microservices?

1. **Use a Single Source of Truth (SSOT):** Designate one microservice or database as the authoritative source for a particular type of data. Other microservices that need this data should query or subscribe to it rather than maintaining their copies. This ensures that there is only one version of truth for the data.
2. **Synchronous Communication:** In cases where strong consistency is crucial, use synchronous communication patterns such as request-response (e.g., HTTP) when accessing critical data. This allows one microservice to wait for a response from another, ensuring that the data is up-to-date before proceeding.
3. **Event-Driven Architecture:** Implement an event-driven architecture using message brokers like Apache Kafka, RabbitMQ, or AWS SNS/SQS. Microservices can publish events when data changes occur, and other services can subscribe to these events to update their local data stores accordingly. Eventual consistency can be acceptable in scenarios where real-time consistency is not mandatory.
4. **CQRS (Command Query Responsibility Segregation):** Separate the responsibility for handling commands (updates) from that of handling queries (reads). This allows you to optimize data storage and access patterns for each, potentially leading to better consistency for both. Event sourcing is often used in conjunction with CQRS to maintain a history of changes.
5. **Saga Pattern:** Implement long-running, multi-step transactions using the saga pattern. A saga is a sequence of local transactions across multiple microservices that ensure data consistency. If a step fails, compensating transactions can be executed to undo the changes made so far.

# How can you implement fault tolerance and resilience in a microservices system?

1. **Use Redundancy:**
   * Deploy multiple instances of critical microservices to ensure redundancy. This can be within the same data center or across multiple data centers or cloud regions.
   * Employ load balancing to distribute traffic evenly among these instances, and set up health checks to detect and route away from unhealthy instances.
2. **Circuit Breaker Pattern:**
   * Implement the circuit breaker pattern to prevent cascading failures. A circuit breaker monitors the health of a service and temporarily blocks requests to that service if it is experiencing issues.
   * Provide a fallback mechanism or default response when the circuit breaker is open to maintain some level of service.
3. **Timeouts and Retries:**
   * Set appropriate timeouts for requests between microservices. Timeouts prevent long waits in case a service is unresponsive.
   * Implement retry mechanisms with exponential backoff to automatically retry failed requests. This can help when transient failures occur.
4. **Bulkheads:**
   * Apply the bulkhead pattern to isolate different parts of your application from each other. For example, allocate a separate thread pool for each microservice, so issues in one service won't affect others.
5. **Graceful Degradation:**
   * Design your microservices to gracefully degrade when certain dependencies are unavailable. For example, if a non-critical service is down, the overall functionality should still work without that service.
6. **Distributed Tracing and Monitoring:**
   * Use distributed tracing tools (e.g., OpenTelemetry, Zipkin) to track the flow of requests across microservices. This helps in diagnosing performance bottlenecks and failures.
   * Implement comprehensive monitoring with alerting. Monitor system metrics, logs, and application-specific metrics to detect and respond to anomalies quickly.
7. **Chaos Engineering:**
   * Conduct chaos engineering experiments to proactively inject failures into your system and observe how it behaves. This helps identify weaknesses and improve resilience.
   * Tools like Chaos Monkey (for Netflix OSS) or Chaos Toolkit can help with controlled chaos engineering experiments.
8. **Isolation and Resource Quotas:**
   * Isolate microservices by limiting their resource consumption (CPU, memory, network) through resource quotas. This prevents one service from monopolizing resources and impacting others.
9. **Immutable Infrastructure:**
   * Build your microservices as immutable artifacts (e.g., Docker containers) and replace instances rather than trying to fix them in-place when issues occur. This reduces the risk of configuration drift.

# What tools and techniques can be used for monitoring and debugging microservices in production?

**Monitoring Tools:**

1. **Prometheus:** An open-source monitoring and alerting toolkit that is widely used in microservices environments. It can collect metrics from various sources and supports querying and alerting.
2. **Grafana:** Often used in conjunction with Prometheus, Grafana provides visualization and dashboarding capabilities for your monitoring data.
3. **Elasticsearch, Logstash, and Kibana (ELK Stack):** Used for log aggregation, analysis, and visualization. It helps you centralize logs from all microservices and perform keyword searches, correlation, and anomaly detection.
4. **Jaeger:** An open-source, end-to-end distributed tracing system that helps you monitor and trace requests as they flow through your microservices architecture.
5. **Zipkin:** Another distributed tracing system that can help with identifying performance bottlenecks and latency issues in microservices interactions.
6. **New Relic and AppDynamics:** Commercial monitoring solutions that offer comprehensive application performance monitoring (APM) features, including code-level insights, transaction tracing, and real-time alerts.
7. **Datadog:** A cloud-based monitoring and analytics platform that provides visibility into your microservices infrastructure, including metrics, traces, and logs.
8. **AWS CloudWatch and Azure Monitor:** Cloud-specific monitoring solutions that offer integration with various AWS and Azure services and provide insights into the health and performance of your microservices on these platforms.

**Debugging Techniques:**

1. **Distributed Tracing:** Utilize distributed tracing tools like Jaeger or Zipkin to trace requests as they traverse multiple microservices. This helps identify performance bottlenecks and pinpoint the source of issues.
2. **Logging:** Implement structured and comprehensive logging in your microservices. Use log levels to distinguish between informational, warning, and error messages. Centralize logs for easier analysis.
3. **Dynamic Debugging:** In some cases, you can attach a debugger to a running microservice instance to diagnose issues in real-time. Container orchestration platforms like Kubernetes support this.
4. **Health Checks:** Implement health check endpoints in your microservices that can be probed to determine the health of the service. Monitoring tools can regularly query these endpoints.
5. **Profiling:** Use profiling tools to analyze the runtime behavior and performance of your microservices. Tools like pprof for Go or YourKit for Java can help identify CPU or memory bottlenecks.
6. **Error Tracking:** Employ error tracking tools like Sentry or Rollbar to capture and track errors in your microservices. These tools provide detailed information about the error, including its occurrence and stack trace.
7. **Chaos Engineering:** Conduct controlled experiments to deliberately introduce failures into your production environment (Chaos Engineering). This helps you uncover vulnerabilities and weaknesses in your microservices architecture.

Geo redundancy

Sharding