**High Level Design**

**CAP Theorem**

It states that that a distributed system can deliver only two of three desired characteristics: consistency, availability, and partition tolerance (the ‘C.” ‘A’ and ‘P’ in CAP).

A distributed system is a network that stores data on more than one node (physical or virtual machines) at the same time. Because all cloud applications are distributed systems, it’s essential to understand the CAP theorem when designing a cloud app so that you can choose a data management system that delivers the characteristics your application needs most.

1. **Consistency**: Consistency means that all clients see the same data at the same time, no matter which node they connect to. For this to happen, whenever data is written to one node, it must be instantly forwarded or replicated to all the other nodes in the system before the write is deemed ‘successful.’
2. **Availability**: Availability means that any client making a request for data gets a response, even if one or more nodes are down. Another way to state this—all working nodes in the distributed system return a valid response for any request, without exception.
3. **Partition tolerance**: A partition is a communications break within a distributed system—a lost or temporarily delayed connection between two nodes. Partition tolerance means that the cluster must continue to work despite any number of communication breakdowns between nodes in the system.

**2 Phase Commit**

Two-phase commit (2PC) is a standardized protocol that ensures atomicity, consistency, isolation and durability (ACID) of a transaction; it is an atomic commitment protocol for distributed systems.

In a distributed system, transactions involve altering data on multiple databases or resource managers, causing the processing to be more complicated since the database has to coordinate the committing or rolling back of changes in a transaction as a self-contained unit; either the entire transaction commits or the entire transaction rolls back.

A transaction manager uses 2PC to ensure data integrity as well as the integrity of the global database -- the collection of databases participating in the transaction -- as well as monitor the commitment or rollback of the distributed transactions. This protocol is entirely transparent and requires no programming by the user or application developer.

**How does two-phase commit work?**

In order for a distributed transaction to take place, a special object, known as a coordinator, is required. The coordinator is in charge or arranging activities and synchronizations between distributed servers.

As the name implies, 2PC consists of two phases:

**Phase 1 (the prepare phase)** - The protocol ensures all resource managers have saved the transaction’s updates to stable storage. Every server that is required to commit writes its data records in a log. If a server is unsuccessful in doing so, then it responds with a failure message; if it is successful, then it sends an OK message.

In this first phase, the initiating node requests all other participating nodes to promise to either commit or roll back the transaction.

There are three types of responses that the responding node can send back:

* **Prepared** - A prepared response is given when data in the node has been revised by a statement in the distributed transaction and the node has successfully composed itself for commitment or rollback. The prepared response also ensures that locks held for the transaction can survive a failure.
* **Read-only** - A read-only response means that data on the node has been queried, but it cannot be modified. Therefore, no preparation is necessary.
* **Abort** - An abort response indicates that the node cannot successfully prepare itself for commitment.

In order for the prepare phase to reach completion and one of the three messages to be sent, each node, except for the commit point site, must perform several steps. First, the node must request that the following referenced nodes are ready to commit. Then the node checks if the transaction changes data on itself or the subsequent nodes. If the data does not change, then the node skips the rest of the steps and replies with the read-only response.

If the data does change, then the node assigns the resources it needs to commit the transaction. The node will save redo records matching the changes made by the transaction to its redo log. A lock is then placed on the modified tables to prevent them from being read.

Next, the node ensures that locks held for the transaction can survive a failure. If all steps go according to plan, then the node issues a prepared response. However, if the attempts of the node, or one of its subsequent nodes, are unsuccessful in preparing to commit, then it issues the abort response.

Prepared nodes then wait for either a commit or rollback response from the global coordinator. The prepared nodes are considered to be in-doubt until all changes are either committed or rolled back.

**Phase 2 (the commit phase)** - If phase one is successful and all participants send an OK response, then phase two tells all resource managers to commit. After committing, each node logs its commit in a record and sends the coordinator a message indicating that its commit was successful. If phase one fails, then phase two tells the resource managers to abort, all servers roll back and each node sends feedback that the rollback has been successfully accomplished.

The commit phase can be broken down into the following steps:

* The global coordinator prompts the commit point site to commit and the action is performed.
* The commit point site records its commitment and sends a response back to the global coordinator, informing that it has successfully committed.
* The global and local coordinators instruct all other nodes to commit to the transaction.
* Each node's database releases its locks and commits its local portion of the distributed transaction.
* Each node's database registers an additional redo entry in its local log to show that is has committed the transaction.
* All participating nodes alert the global coordinator to the status of their successful commitment.

Once the commit phase is complete, all nodes in the distributed system possess consistent data.

In a distributed system, databases can independently fail and recover. As a result, it’s possible for a transaction to successfully commit its updates on one database system, but not on another due to a system failure. When the failed database recovers, it must be able to commit the transaction. To do so, the system must have a copy of the transaction’s updates that were executed there. However, when a system fails, it can lose the contents of its main memory. The database must therefore store a copy of the transaction’s updates before a failure occurs. 2PC ensures that each system accessed by a transaction durably stores its portion of the transaction’s updates before the transaction commits anywhere.

2PC is usually implemented by a transaction manager. The transaction manager tracks which resource managers are accessed by each transaction and runs the 2PC protocol.

**Strangler Pattern**

**Definition**: Incrementally migrating a legacy system by gradually replacing specific pieces of functionality with new applications and services. As features from the legacy system are replaced, the new system eventually replaces all of the old system's features, strangling the old system and allowing you to decommission it.

**Context and problem**: As systems age, the development tools, hosting technology, and even system architectures they were built on can become increasingly obsolete. As new features and functionality are added, the complexity of these applications can increase dramatically, making them harder to maintain or add new features to. Completely replacing a complex system can be a huge undertaking. Often, you will need a gradual migration to a new system, while keeping the old system to handle features that haven't been migrated yet. However, running two separate versions of an application means that clients have to know where particular features are located. Every time a feature or service is migrated, clients need to be updated to point to the new location.

**Solution:** Incrementally replace specific pieces of functionality with new applications and services. Create a façade that intercepts requests going to the backend legacy system. The façade routes these requests either to the legacy application or the new services. Existing features can be migrated to the new system gradually, and consumers can continue using the same interface, unaware that any migration has taken place. This pattern helps to minimize risk from the migration, and spread the development effort over time. With the façade safely routing users to the correct application, you can add functionality to the new system at whatever pace you like, while ensuring the legacy application continues to function. Over time, as features are migrated to the new system, the legacy system is eventually "strangled" and is no longer necessary. Once this process is complete, the legacy system can safely be retired.

Source: https://learn.microsoft.com/en-us/azure/architecture/patterns/strangler-fig

**Saga Pattern**

The Saga design pattern is a way to manage data consistency across microservices in distributed transaction scenarios. A saga is a sequence of transactions that updates each service and publishes a message or event to trigger the next transaction step. If a step fails, the saga executes compensating transactions that counteract the preceding transactions.

**Context and problem**: A transaction is a single unit of logic or work, sometimes made up of multiple operations. Within a transaction, an event is a state change that occurs to an entity, and a command encapsulates all information needed to perform an action or trigger a later event. Transactions must be atomic, consistent, isolated, and durable (ACID). Transactions within a single service are ACID, but cross-service data consistency requires a cross-service transaction management strategy.

In multiservices architectures:

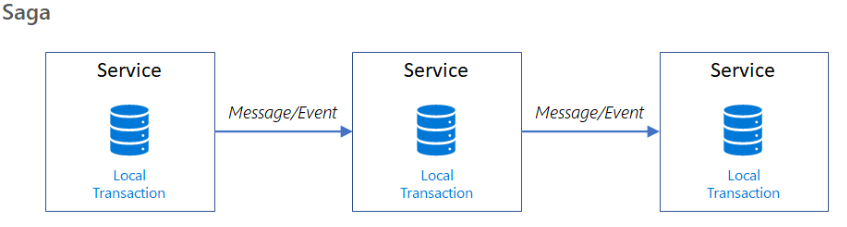
* Atomicity is an indivisible and irreducible set of operations that must all occur or none occur.
* Consistency means the transaction brings the data only from one valid state to another valid state.
* Isolation guarantees that concurrent transactions produce the same data state that sequentially executed transactions would have produced.
* Durability ensures that committed transactions remain committed even in case of system failure or power outage.

A database-per-microservice model provides many benefits for microservices architectures. Encapsulating domain data lets each service use its best data store type and schema, scale its own data store as necessary, and be insulated from other services' failures. However, ensuring data consistency across service-specific databases poses challenges.

Distributed transactions like the two-phase commit (2PC) protocol require all participants in a transaction to commit or roll back before the transaction can proceed. However some participant implementations, such as NoSQL databases and message brokering, don't support this model.

Another distributed transaction limitation is interprocess communication (IPC) synchronicity and availability. Operating system-provided IPC allows separate processes to share data. For distributed transactions to commit, all participating services must be available, potentially reducing overall system availability. Architectural implementations with IPC or transaction limitations are candidates for the Saga pattern.

**Solution**: The Saga pattern provides transaction management using a sequence of local transactions. A local transaction is the atomic work effort performed by a saga participant. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails, the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.



In Saga patterns:

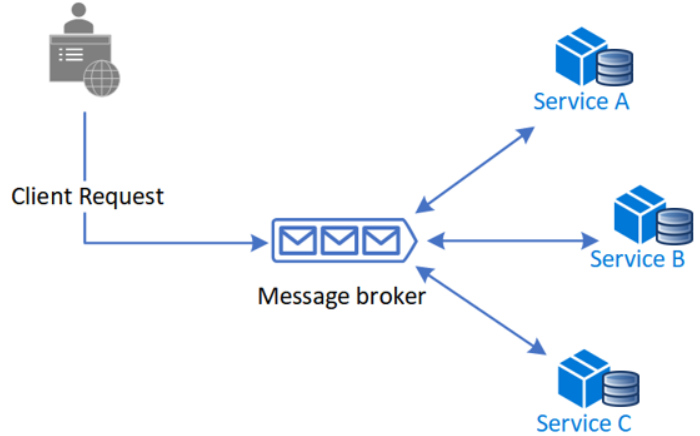
* Compensable transactions are transactions that can potentially be reversed by processing another transaction with the opposite effect.
* A pivot transaction is the go/no-go point in a saga. If the pivot transaction commits, the saga runs until completion. A pivot transaction can be a transaction that is neither compensable nor retryable, or it can be the last compensable transaction or the first retryable transaction in the saga.
* Retryable transactions are transactions that follow the pivot transaction and are guaranteed to succeed.

There are two common saga implementation approaches, choreography and orchestration. Each approach has its own set of challenges and technologies to coordinate the workflow.

* Choreography
* Orchestrator

**Choreography**

Choreography is a way to coordinate sagas where participants exchange events without a centralized point of control. With choreography, each local transaction publishes domain events that trigger local transactions in other services.



**Benefits**

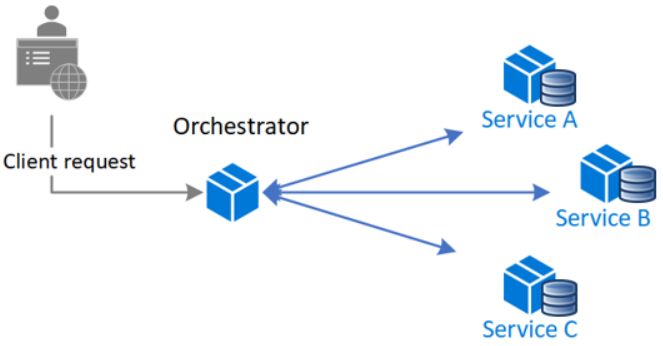
* Good for simple workflows that require few participants and don't need a coordination logic.
* Doesn't require additional service implementation and maintenance.
* Doesn't introduce a single point of failure, since the responsibilities are distributed across the saga participants.

**Drawbacks**

* Workflow can become confusing when adding new steps, as it's difficult to track which saga participants listen to which commands.
* There's a risk of cyclic dependency between saga participants because they have to consume each other's commands.
* Integration testing is difficult because all services must be running to simulate a transaction.

**Orchestration**

Orchestration is a way to coordinate sagas where a centralized controller tells the saga participants what local transactions to execute. The saga orchestrator handles all the transactions and tells the participants which operation to perform based on events. The orchestrator executes saga requests, stores and interprets the states of each task, and handles failure recovery with compensating transactions.



**Benefits**

* Good for complex workflows involving many participants or new participants added over time.
* Suitable when there is control over every participant in the process, and control over the flow of activities.
* Doesn't introduce cyclical dependencies, because the orchestrator unilaterally depends on the saga participants.
* Saga participants don't need to know about commands for other participants. Clear separation of concerns simplifies business logic.

**Drawbacks**

* Additional design complexity requires an implementation of a coordination logic.
* There's an additional point of failure, because the orchestrator manages the complete workflow.

**Issues and considerations**

Consider the following points when implementing the Saga pattern:

* The Saga pattern may initially be challenging, as it requires a new way of thinking on how to coordinate a transaction and maintain data consistency for a business process spanning multiple microservices.
* The Saga pattern is particularly hard to debug, and the complexity grows as participants increase.
* Data can't be rolled back, because saga participants commit changes to their local databases.
* The implementation must be capable of handling a set of potential transient failures, and provide idempotence for reducing side-effects and ensuring data consistency. Idempotence means that the same operation can be repeated multiple times without changing the initial result. For more information, see the guidance on ensuring idempotence when processing messages and updating state together.
* It's best to implement observability to monitor and track the saga workflow.
* The lack of participant data isolation imposes durability challenges. The saga implementation must include countermeasures to reduce anomalies.

The following anomalies can happen without proper measures:

* *Lost updates*, when one saga writes without reading changes made by another saga.
* *Dirty reads*, when a transaction or a saga reads updates made by a saga that has not yet completed those updates.
* *Fuzzy/nonrepeatable reads*, when different saga steps read different data because a data update occurs between the reads.

Suggested countermeasures to reduce or prevent anomalies include:

* *Semantic lock*, an application-level lock where a saga's compensable transaction uses a semaphore to indicate an update is in progress.
* *Commutative updates* that can be executed in any order and produce the same result.
* *Pessimistic view*: It's possible for one saga to read dirty data, while another saga is running a compensable transaction to roll back the operation. Pessimistic view reorders the saga so the underlying data updates in a retryable transaction, which eliminates the possibility of a dirty read.
* *Reread value* verifies that data is unchanged, and then updates the record. If the record has changed, the steps abort and the saga may restart.
* A *version file* records the operations on a record as they arrive, and then executes them in the correct order.
* By *value* uses each request's business risk to dynamically select the concurrency mechanism. Low-risk requests favor sagas, while high-risk requests favor distributed transactions.

**Two-phase commit vs. Saga**

Sagas and 2PC have the same goal: to coordinate resources while overlaying operations form a coherent unit of work. As a result, both protocols will produce a consistent system state at the end. However, the two protocols utilize different approaches to reach this goal. Specifically, Saga uses units of work that can be unfinished; a commitment protocol is not included.

The Saga pattern is a sequence of local transactions in which each transaction modifies data within a single service. Unlike 2PC, which waits for all nodes to be ready to commit or rollback before performing the action, Sagas individually respond to an external request matching the system operation and then triggers each subsequent step with the completion of the proceeding one.

Also, changes made by Saga operations are immediately visible to the outside world. This is because Saga instantly commits resource-located transactions after each step in the business process ends. On the other hand, 2PC resource-located transactions carry through just about the whole global transaction lifetime.

As a result, 2PC allows programmers to commit the entire transaction in one request with this request spanning over various systems and networks. If each participating system and network abides by the protocol, then the entire transaction can easily commit or rollback.

Saga allows programmers to split the transaction into multiple steps, allowing the protocol to span extensive periods or time, but not necessarily over systems and networks. Consequently, 2PC is used for more immediate transactions while Saga is utilized in long running transactions.

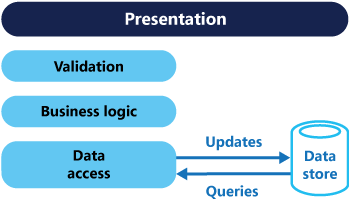
2PC is also easier for the application programmer to use since the responsibility of managing all the transaction troubleshooting falls on the transaction manager. This means the programmer only has to be concerned with their business logic -- such as inserting data into the database or sending a message to the queue. Sagas require an extra step for the programmer because the protocol requires a compensating action to be created and defined for each specific Saga pattern.

**CQRS Pattern**

CQRS stands for Command and Query Responsibility Segregation, a pattern that separates read and update operations for a data store. Implementing CQRS in your application can maximize its performance, scalability, and security. The flexibility created by migrating to CQRS allows a system to better evolve over time and prevents update commands from causing merge conflicts at the domain level.

**Context and problem**

In traditional architectures, the same data model is used to query and update a database. That's simple and works well for basic CRUD operations. In more complex applications, however, this approach can become unwieldy. For example, on the read side, the application may perform many different queries, returning data transfer objects (DTOs) with different shapes. Object mapping can become complicated. On the write side, the model may implement complex validation and business logic. As a result, you can end up with an overly complex model that does too much.



Read and write workloads are often asymmetrical, with very different performance and scale requirements.

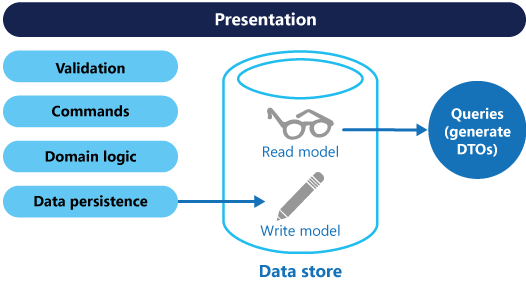
* There is often a mismatch between the read and write representations of the data, such as additional columns or properties that must be updated correctly even though they aren't required as part of an operation.
* Data contention can occur when operations are performed in parallel on the same set of data.
* The traditional approach can have a negative effect on performance due to load on the data store and data access layer, and the complexity of queries required to retrieve information.
* Managing security and permissions can become complex, because each entity is subject to both read and write operations, which might expose data in the wrong context.

**Solution**

CQRS separates reads and writes into different models, using commands to update data, and queries to read data.

* Commands should be task-based, rather than data centric. ("Book hotel room", not "set ReservationStatus to Reserved").
* Commands may be placed on a queue for asynchronous processing, rather than being processed synchronously.
* Queries never modify the database. A query returns a DTO that does not encapsulate any domain knowledge.

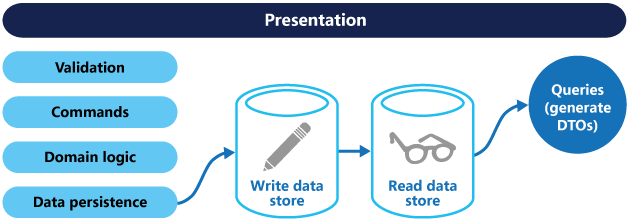
The models can then be isolated, as shown in the following diagram, although that's not an absolute requirement.



Having separate query and update models simplifies the design and implementation. However, one disadvantage is that CQRS code can't automatically be generated from a database schema using scaffolding mechanisms such as O/RM tools (However, you will be able to build your customization on top of the generated code).

For greater isolation, you can physically separate the read data from the write data. In that case, the read database can use its own data schema that is optimized for queries. For example, it can store a materialized view of the data, in order to avoid complex joins or complex O/RM mappings. It might even use a different type of data store. For example, the write database might be relational, while the read database is a document database.

If separate read and write databases are used, they must be kept in sync. Typically this is accomplished by having the write model publish an event whenever it updates the database. For more information about using events, see Event-driven architecture style. Since message brokers and databases usually cannot be enlisted into a single distributed transaction, there can be challenges in guaranteeing consistency when updating the database and publishing events. For more information, see the guidance on idempotent message processing.



The read store can be a read-only replica of the write store, or the read and write stores can have a different structure altogether. Using multiple read-only replicas can increase query performance, especially in distributed scenarios where read-only replicas are located close to the application instances.

Separation of the read and write stores also allows each to be scaled appropriately to match the load. For example, read stores typically encounter a much higher load than write stores.

**Benefits of CQRS**

* *Independent scaling*. CQRS allows the read and write workloads to scale independently, and may result in fewer lock contentions.
* *Optimized data schemas***.** The read side can use a schema that is optimized for queries, while the write side uses a schema that is optimized for updates.
* *Security***.** It's easier to ensure that only the right domain entities are performing writes on the data.
* *Separation of concerns*. Segregating the read and write sides can result in models that are more maintainable and flexible. Most of the complex business logic goes into the write model. The read model can be relatively simple.
* *Simpler queries*. By storing a materialized view in the read database, the application can avoid complex joins when querying.

**Challenges of implementing CQRS**

* **Complexity**. The basic idea of CQRS is simple. But it can lead to a more complex application design, especially if they include the Event Sourcing pattern.
* **Messaging**. Although CQRS does not require messaging, it's common to use messaging to process commands and publish update events. In that case, the application must handle message failures or duplicate messages. See the guidance on Priority Queues for dealing with commands having different priorities.
* **Eventual consistency**. If you separate the read and write databases, the read data may be stale. The read model store must be updated to reflect changes to the write model store, and it can be difficult to detect when a user has issued a request based on stale read data.

Low Level Design