WSO2

White Paper

Transactions in a Microservice World

By

Frank Leymann

Consultant

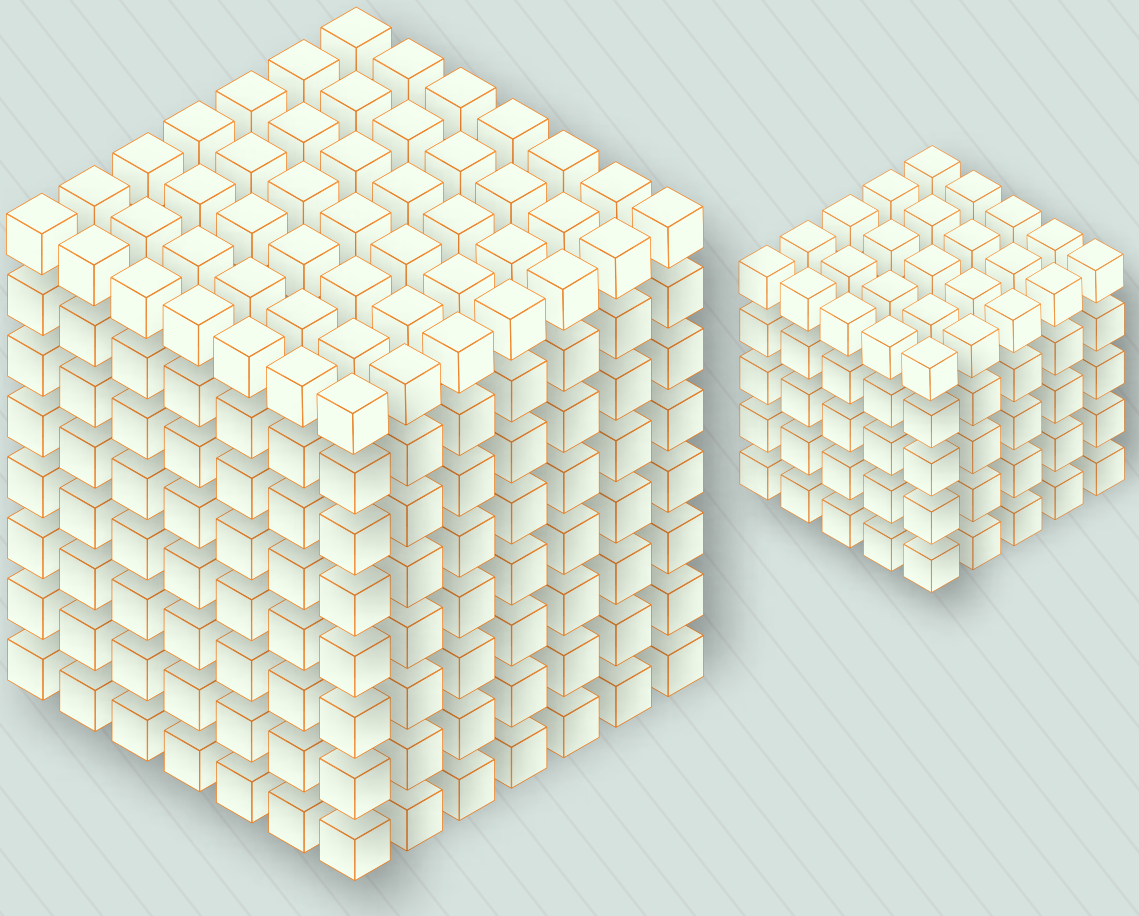

Table of Contents

Abstract

Typically, microservice-based applications distribute data widely, especially in cloud-based applications, resulting in distributed applications. This impacts the transactions within these applications. This white paper refreshes the concepts of classical and distributed transactions. Next, we explain how cloud-based applications are affected by distribution. Finally, we present compensation-based transactions as a reliable method for microservice-based application transactions, even in the cloud.

1. Classical Transactions

We will briefly overview the evolution of the transaction concept, starting by summarizing the ACID paradigm, which the classical world considers synonymous with “transaction”. We will also present a classical example of a transaction that will serve as a running example throughout the white paper and will be refined in subsequent sections.

1.1 Origin

The sphere of control [1] is often considered a precursor to what has later become known as a transaction in the context of databases [2] (sometimes also referred to as a logical unit of work). A *sphere of control* prevents others from modifying the information processed within it while it is still active. It also determines which steps share the same identity, enabling the tracking of modifications. Additionally, it controls the use of its results and can undo its processing unilaterally. The transaction concept [3] builds upon the sphere of control by defining a transaction as a collection of data transformations that possess the properties of atomicity, durability, and consistency (see section 1.2). These properties ensure that a transaction is either completed entirely or not at all, that its effects are persistent, and that the database remains in a consistent state.

When a transaction is successful, all modifications made by its steps become visible to the outside world. However, if an error occurs during transaction execution, the modifications made by its steps are undone, and the world’s state is restored to its original state. This principle is fundamental to the original transaction concept, ensuring the effects of a transaction are either fully applied or fully rolled back. However, this principle cannot be guaranteed in compensation-based recovery (discussed in section 4.3).

1.2 The ACID Paradigm

The properties of a transaction are atomicity, consistency, isolation, and durability, forming the mnemonic acronym ACID [4]. We briefly define these properties as follows (for more details see [5]):

- *Atomicity* (A): A transaction must either fully apply or fully rollback all steps within it to ensure atomicity.
- *Consistency* (C): To guarantee consistency, a transaction must transform from one consistent state into another consistent state, where a state is considered consistent if it adheres to a set

of rules associated with the application domain. Throughout the transaction execution, this property guarantees that the database remains in a valid state.

- *Isolation* (I): To ensure data integrity and prevent data corruption, a transaction's effects must be hidden from other transactions during its execution, which is achieved by keeping transactions isolated.
- *Durability* (D): To ensure persistence of a transaction's efforts and allow for recovery in case of a system malfunction, the results of a successful transaction must be guaranteed to survive any malfunctions.

Researchers have developed a vast body of knowledge and techniques to implement systems that adhere to the ACID paradigm [6]. Although the referenced book may be considered old, it remains the ultimate reference for a background on classical transactions and distributed transactions. Various techniques like concurrency control mechanisms based on locking and versioning are used to implement the ACID paradigm. These techniques ensure that transactions execute in isolation and maintain data consistency throughout the transaction's lifespan. These techniques can be applied in the context of microservices, as discussed in section 3.4.

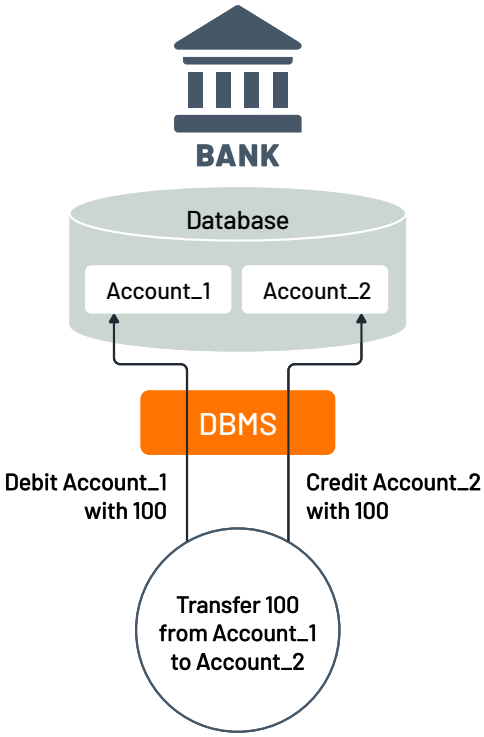

Figure 1: The classical funds transfer

Figure 1 shows a funds transfer, a common example of a transaction. Suppose there are two accounts, Account_1 and Account_2, within the same bank database. A microservice function is needed to transfer 100 currency units from Account_1 to Account_2. The microservice implements the transfer as a transaction to make this a reality. It starts the transaction using an explicit operation

like BEGIN, credits 100 currency units to Account_2, debits 100 currency units from Account_1, and finally ends the transaction. If no errors are detected,, the microservice finalizes the transaction with COMMIT, requesting the environment to ensure the atomicity of the transaction and the durability of its results, while ensuring the isolation of the transaction. The microservice's logic guarantees consistency by ensuring the amount credited is the same as the amount debited, as required by the rules of a funds transfer. However, if an error is detected during processing (i.e., if the balance of Account_1 is less than 100 currency units), the transaction ends with ROLLBACK, requesting the environment to undo the credit and debit steps. The environment ensures isolation and atomicity as well. Explicitly beginning and ending a transaction is called transaction bracketing, which is considered a best practice.

The environment can unilaterally decide to undo a transaction, even if the microservice asks to COMMIT it, if it identifies erroneous situations, such as the inability to guarantee durability of the results.

2. Distributed Transactions

Companies often use multiple storage systems, resulting in distributed transactions that pose the challenge of atomic commitment. The two-phase-commit protocol is a possible solution to address this issue. To illustrate this, let us consider an example of a distributed transaction based on microservices. Additionally, data replication is another form of distribution that is worth discussing. Resource managers often employ data replication to implicitly exploit replication.

2.1 Several Storage Systems

In practice, companies often manage their data using multiple storage systems for several reasons:

- A single type of database may not be appropriate for a company due to data diversity. For example, tabular data is suitable for relational database systems, while highly networked data is best managed by graph databases.
- Sometimes, companies use multiple database systems of the same type, even if the data is homogeneous. For instance, they may use multiple relational database systems from the same vendor for workload distribution or disaster recovery preparation. Alternatively, companies may use relational database systems from different vendors in different locations due to local buying preferences or as part of a vendor diversity strategy.
- Some data has unique characteristics that require specialized storage systems. For example, messages often require durability for availability reasons but only for a short period of time

(known as “in-flight” data). To manage this type of data, separate message stores are used to manage specialized stores like queues and topics.

- The technology autonomy of microservices is a key benefit, allowing each microservice to independently select the storage technology that is most suitable for implementing its specific functionality.

Ensuring message integrity requires performing multiple data manipulations in a single transaction when dealing with data from multiple storage systems. For instance, when a request message in a queue needs to manipulate data in a relational database and generate a response message to be sent to another queue, the entire process of receiving the request message, manipulating the tables, and submitting the response message must be atomic and isolated. This necessitates performing the overall processing within a transaction.

Enabling the overall processing to occur within a transaction requires that all relevant storage systems support transactions and participate in a joint transaction. Storage systems that are capable of participating in joint transactions are called *resource managers*. Additionally, functions in an application that involve resource managers from different organizations may occur. Transactions that involve several resource managers distributed across different locations are known as *distributed transactions*.

The ACID paradigm applies to distributed transactions as well, which means that each resource manager must conduct atomic, consistent, isolated, and durable data manipulations. These data manipulations that affect a particular resource manager within a distributed transaction are referred to as *local transactions*. However, local transactions within a distributed transaction may not necessarily be executed sequentially. Instead, they can be intertwined, meaning that a data manipulation of one local transaction may occur between two data manipulations of another local transaction. It is even possible for data manipulations of different local transactions to occur simultaneously.

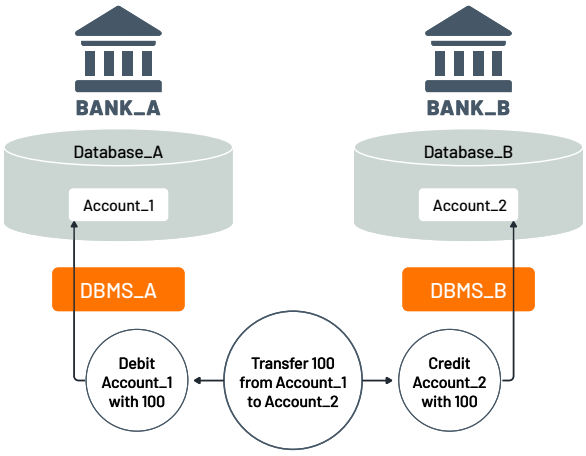

Figure 2: Funds transfer with a distributed transaction

Figure 2 illustrates a practical example of a funds transfer using a distributed transaction. Say

Account_1 is held in Bank_A, with the account data stored in Database_A managed by a database management system DBMS_A, while Account_2 is held in Bank_B, its data is stored in Database_B, and is managed by DBMS_B. A distributed transaction is used to execute the funds transfer, which involves a debit function provided by Bank_A and a credit function provided by Bank_B. The funds transfer can be done using microservices, where a transfer microservice invokes debit and credit microservices. A debit microservice decreases an account by a specific amount, whereas a credit microservice increases an account by a specific amount. In turn, the debit and credit microservices use native data manipulation functions of the corresponding database management systems to modify the account data. These data manipulations are executed as a local transaction on DBMS_A and DBMS_B, respectively. Both local transactions represent the overall distributed transaction of the fund transfer.

2.2 Two-Phase-Commit

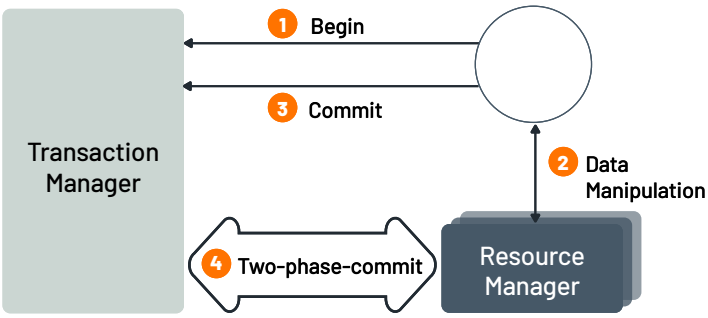

Figure 3: Two-phase-commit in a nutshell

The fundamental issue known as *atomic commitment* is encountered in distributed transactions. Simply asking each of the resource managers is insufficient for committing the overall distributed transaction. This is because the outcome of the commit processing may not be atomic. For instance, if three resource managers are involved and a commit is sent to all of them, the result may be that two resource managers successfully process the commit, while the third rolls back because it detected an error during commit processing. A resource manager may unilaterally decide to undo a transaction at any time before signaling successful completion of commit processing.

A protocol like the well-known two-phase-commit protocol [8] is required to achieve atomic commitment [7] in distributed transactions. The protocol operates in two phases (hence, the name) controlled by a special middleware known as a *transaction manager (also known as a coordinator)*. As seen by Figure 3, the program implementing the distributed transaction utilizes the transaction manager to begin the transaction and to end it by committing or rolling back the transaction. The

transaction manager runs the first phase, the voting phase, by requesting each participating resource manager to prepare itself to commit its local transaction when the program wants to commit the transaction. Each resource manager responds whether it can guarantee to commit its local transaction. The transaction manager decides on the processing for the second phase, the completion phase, once all the responses have been received. If all resource managers are guaranteed to commit, the transaction manager instructs each resource manager to do so; otherwise, the transaction manager instructs the resource managers to rollback. This ensures that atomic commitment is achieved (for more details, see [5]). Several open source implementations of transaction managers are available, such as Seata [9], Atomicos [10], or Narayana [11].

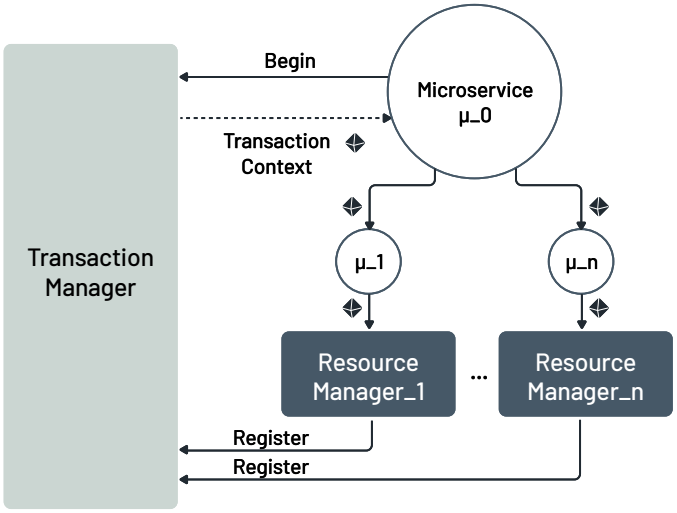

Figure 4: Passing transaction context

It is very helpful to understand how different requests in a distributed transaction work together. Figure 4 shows a microservice called μ_0 that uses other microservices μ_1 through μ_n, to do its work. The transaction is commenced by μ_0 asking the transaction manager to start it. The transaction manager sends back a *transaction context*, with a unique identifier for the distributed transaction and its own address. When μ_0 asks μ_j to do something, it sends the transaction context with the request (as part of the message header of the request message). μ_j realizes it is part of a distributed transaction when it sees the transaction context and cannot decide the transaction's outcome anymore. It is now *infected* by the distributed transaction. When μ_j asks its resource manager to make changes, it also sends the transaction request along. The resource manager sees the transaction request and realizes it is infected too.

When processing the first manipulation request of μ_j, the resource manager gets the address of the transaction manager from the context and sends its own address to be used in the two-phase-commit protocol. Now, resource manager μ_j is controlled by the transaction manager for the current distributed transaction and can only end its own local transaction under the transaction manager's control. In practice, several factors can slow down distributed transactions. The infection process, the two-phase-commit protocol, network delays, and other reasons all add up. Because of this, it is usually

best to limit the number of resource managers (and microservices) that participate in a distributed transaction. On average, two or three resource managers should be fine. It is important to remember this when deciding to use ACID-based distributed transactions in practice. Other types of distributed transactions can handle more microservices at once (please see section 4 for more details).

2.3 Data Replication

Applications that use microservices are usually distributed. This is especially true in cloud environments, where the microservices that make up an application can be spread out across different locations, far from each other. It is possible that a microservice will not be closed to the data it needs to work properly. Network problems and fragmentations can cause a microservice to be unable to access the data it needs. In cloud environments, the hardware used is often standard, which means it can fail. If this occurs, the data stored on that hardware becomes unavailable. Data in cloud environments are often copied or replicated to avoid problems like this. This way, if one copy becomes unavailable, there are still others that can be used.

Data replication means making copies of the same data in different locations [7]. Whenever a change is made in one copy, that change is sent to all the other copies. This can occur immediately, with a delay, or when several changes are sent at once. The user of the data is usually unaware of the replication process. Special middleware is responsible for making and synchnosing copies of the data to ensure they contain the same data.

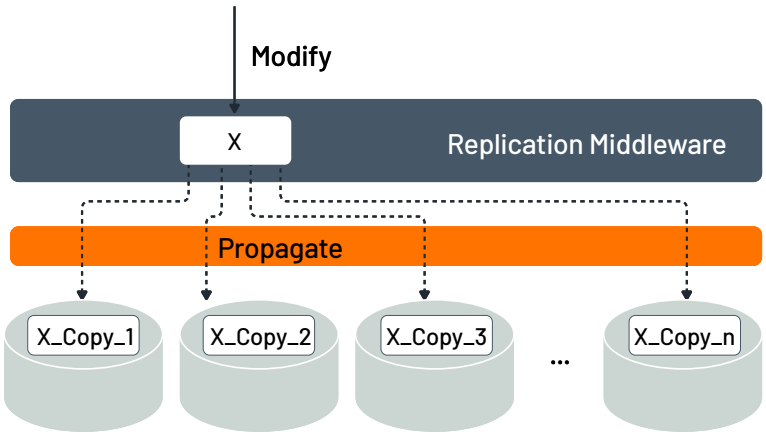

Figure 5: Replicated data at different locations

In Figure 5, a data item called X is copied n times and stored in n different locations. Whenever X is changed, the middleware in charge of the copies of X sends the update to each copy of X. Each copy is referred to as X_Copy_i. How the changes are sent - whether it is right away, later, one at a time, or all at once is irrelevant to the individual making the change.

Data replication has a couple of advantages. Here are some examples:

- Having multiple copies of the same data increases its availability. As long as at least one copy is still accessible, you can manipulate the data, including reading it. If some copies are unavailable, they will eventually become available again and will be updated. This helps prevent network issues or hardware failures from causing problems.
- Several copies help improve response times. When a manipulation request is made, the copy physically closest to the requester, for example, a local one may be used. This reduces the time taken to send the request over the network, making the overall data manipulation faster.
- The danger of data loss is reduced. If one copy is lost, say due to a system crash, you can recover it from one of the remaining copies.

But data replication has its disadvantages too.

- Data replication requires additional effort to distribute copies of the data to target locations.
- Additional processing is needed to maintain consistency of data (please see section 3.2 for more details).
- Data replication increases the amount of storage required to keep the copies of the data.

In an ideal scenario, all replicas of a data item should be modified simultaneously, or at least within a single distributed transaction, to ensure that they always have the same value. The processing overhead (as mentioned in section 2.2) involved in this approach is only tolerable when very few copies of a data item are used. In cloud environments, synchronous data replication is not always practical due to network fragmentation that can hinder the propagation of changes to all copies of a data item. Therefore, cloud environments tend to use asynchronous replication mechanisms, which introduce new consistency models that must be considered by applications. Section 3.2 delves deeper into these consistency models.

2.4 Implicit Use of Replication

The resource manager of middleware controlled by the resource manager typically handles replication. The application may not be unaware of the replication techniques in use. However, one cannot ignore the impact of replication om the application. Replication involves distributing copies of the data, maintaining consistency, and increasing storage requirements. Furthermore, the resulting consistency model deeply impacts the application design. Therefore, designers should keep replication in mind when creating applications to ensure that they can handle the consistency model and benefit from replication. In section 3, we will explore the impact of consistency models on application design.

Figure 6 shows a scenario where a database management system (DBMS) consists of multiple database systems named DBMS_1 to DBMS_n. Whenever any instance manages data, it is

automatically copied to all other instances. Therefore, a modification request from an application to any instance will change all other instances without the application's notice.As a result, the DBMS looks like a single virtual system that stores data in several locations. This allows the data to be accessible in different parts of a cloud provider.

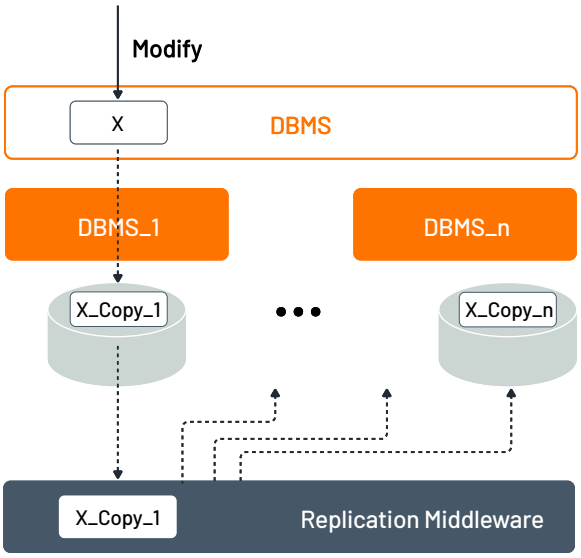

Figure 6: Replication in distributed databases

3. Highly Distributed Transactions

Various autonomous components, each using its own underlying technology, including the database technology, make up microservice-based applications, which are highly distributed. As discussed in section 2.4, several resource managers, which may use replication, must be handled by the entire application. A microservice-based application's components often become geographically dispersed when it runs in a cloud environment. Broken network connections and server outages can occur, resulting in errors that must be addressed by the application, as noted in [12].

If a microservice-based application in the cloud needs transaction support, the transaction will become highly distributed. If an individual microservice uses transactions on a single resource manager that is not part separate from any other microservice's processing, classical transactions can still be achieved, even if the underlying resource manager relies on replicated data.

3.1 CAP

An ideal distributed application should ensure a consistent view of its underlying data and provide all

its functions on each responding node that hosts the application, even if not all nodes hosting pieces of the application are available. Therefore, an ideal distributed application should have the following features:

- *Consistency* (C) is when all copies of a data item have the same value, enabling any responding processing node to read the actual value of a data item.
- *Availability* (A) denotes that data is available for updates at any responding processing node, ensuring that failures do not affect the set of functions offered by the overall application.
- *Partition-tolerance* (P) assumes that failures of network connections or processing nodes do not affect the overall application's functionality.

Unfortunately, in general, an application cannot have all three CAP features simultaneously. This was first suggested in [13] but was precisely formulated and proved later in [14]. This is now known as the *CAP theorem*, which states that "one has to choose two out of three".

For highly-distributed applications, it is assumed that partitions must be tolerated [15]. Therefore, such applications must choose between availability and consistently at any given time. Choosing consistency means that the application can be both consistent and available if it is known in a certain context or situation that no partitioning can occur, in addition to being partition-tolerant.

An application may choose to support different pairs of features at different times. Additionally, different parts of an overall application may make different choices. For instance, if one part of an application runs on the same server, it may be both consistent and available in case no partitioning needs to be tolerated, as it cannot occur. Such considerations have led to a revision [15] of the CAP theorem.

3.2 Consistency Models

In the context of replicated data, ensuring consistency involves guaranteeing that accessing any replica of a data item after a modification request will return the same result. This assumes that all replicas are updated simultaneously and differs significantly from the notion of consistency in the ACID paradigm. In the latter, consistency refers to a set of rules that the collection of data manipulations of a transaction must adhere to (see section 1.2), but it makes no statement about the timeliness of the manipulations.

Over time, the concept of consistency in replicated data has evolved into a non-binary concept with several weaker forms of consistency being defined and implemented in various systems. The strongest form of consistency, known as *strict consistency*, ensures that all replicas have the same value. Weaker forms of consistency that have been defined, like *monotonic read* which guarantees that subsequent access to a data item will never return any previous values to the client once a client has seen a particular value. Similarly, *read-your-writes* guarantees that a client will always access the

updated value of a data item after updating it and never see an older value. These weaker forms of consistency are collectively known as *weak consistency*.

Eventual consistency is a weaker variant of consistency in the context of replicated data, which guarantees that all replicas will eventually converge to the same value for a particular data item, provided no errors occur during replication and no further modifications are made for a sufficiently long period. The duration of the *inconsistency window,* or the period during which different replicas may have different values, depends on various factors such as network delays and the number of replicas.

In distributed systems, conflict detection and resolution is an important aspect of replicated data management. Timestamps are commonly used for detecting conflicts, where each update to a data item is assigned a unique timestamp. When conflicting updates are detected, a reconciliation mechanism is used to determine a final value of concurrent modifications. One common reconciliation policy is “last writer wins”, where the most recent update to a data item is considered the final value. The process of using a set of techniques to determine a final value of concurrent modifications is called *reconciliation*.

Eventual consistency may seem to have limited applicability from an ACID perspective, but many applications can function with inconsistent data to some extent in practice. For instance, in flight reservations, where the probability of a collision between concurrent transactions is low, eventual consistency may be a suitable consistency model. If a reservation is made on a flight and another reservation takes place within the inconsistency window, the flight will be overbooked. In such cases, compensation actions like offering an incentive to someone to take a later flight can be taken to resolve the inconsistency, as discussed in section 4.3.

3.3 BASE

Replicated data imposes consistency models that make ACID-based transactions impractical. Instead, transactions based on the BASE paradigm are used, which stands for “basically available”, “soft-state”, and “eventual consistency”. The definition of BASE is not as clear-cut as that of ACID or CAP, so the following explanation may be somewhat subjective. Nonetheless, in this context, the terms can be understood as follows, according to sources [16, 17].

- *Basically available* (BA) means the application can handle some failures. When a node fails, its functions and data become unusable, but the remaining functions and data are available. If the data is copied between failed nodes and responding nodes, the data may not be in sync and require reconciliation once the failures are recovered.
- *Soft-state* (S) is characterized by changes made to the data being spread throughout the environment after an application finishes. This means that the required updates might not have been applied to the target stores, including updates to replicas and “in-flight” requests that are

currently being processed for further modifications (as explained below).

- *Eventual consistency* (E) is defined as before.

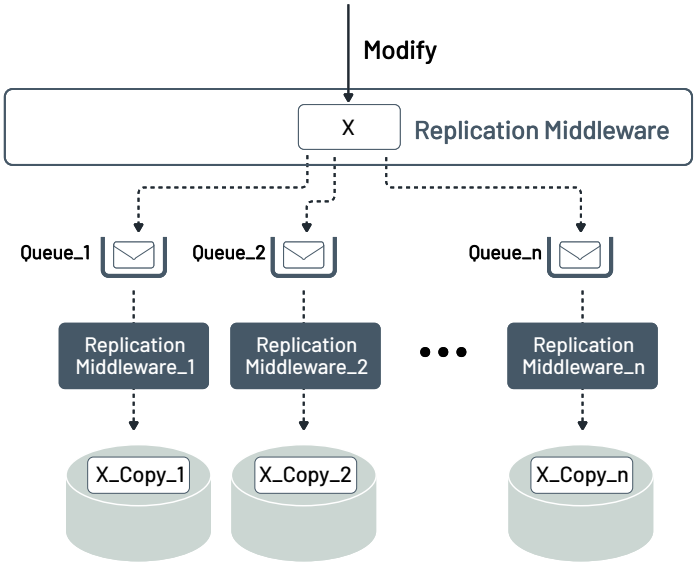

Figure 7: Soft-state resulting from lazy replication

Let us break down the soft-state concept further. Figure 7 illustrates where a data item called X is replicated across multiple locations. When a modification for X is requested, the replication middleware (illustrated by the top dashed line in Figure 7) receives the request and places it as a message into a message queue. The queue serves as the input queue for each local replication component. These replication components then receive the update requests from their message queue and update their local replicas accordingly. However, since not all local replication components will process their update requests simultaneously, there is a time window in which some replicas will have the new value while others will still hold the old value. But the new value for the stale replicas is in the corresponding message queues, and these stale replicas will eventually be updated, achieving eventual consistency.

This way, the overall state of all replicas is “soft,” with the new values spread across the data stores for those data items already updated, and the message queues hold the new value in the form of update requests for data items not yet updated. It is important to note that information that is intended to be made persistent in a data store but has not yet been hardened is called “in-flight information.” In this terminology, the queued update requests are referred to as in-flight requests.

Figure 8 shows the use of the BASE paradigm in the implementation of a fund transfer between two accounts. As before, Account_1 is managed by Bank_A in Database_A of DBMS_A, while Account_2 is managed by Bank_B in Database_B of DBMS_B. The transfer is implemented using three microservices: a debit microservice that reduces an account by a given amount, a credit microservice that increases an account by a given amount, and a transfer microservice that uses the debit and

credit microservices to implement the transfer in a mixture of synchronous and asynchronous requests. If the funds transfer is from Account_1 to Account_2, the transfer microservice immediately reduces the amount of Account_1 by synchronously invoking the debit microservice. It also puts a corresponding credit request as a message into the input queue of the credit microservice. The credit microservice processes the credit request message from its input queue at any convenient time, meaning that the increase of Account_2 may take place later. Thus, during the time when the credit request has not yet been processed, the amounts of the affected accounts do not reflect the funds transfer. However, the in-flight credit request ensures that eventually both accounts will reflect the semantics of the fund transfer, achieving eventual consistency. In this way, the overall state of the fund transfer is “soft” during that time period.

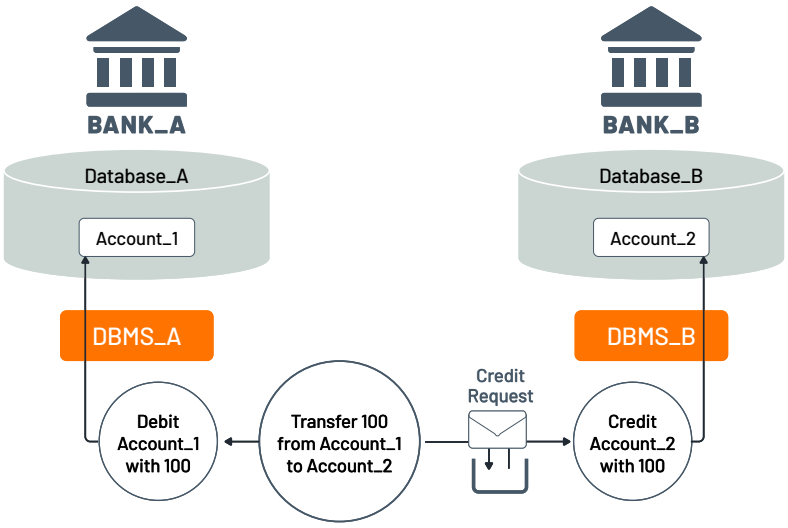

Figure 8: Funds transfer as a BASE transaction

Placing the credit request message in the queue is also a synchronous action. Once the message is delivered to the queue, the put request is complete. Note that both the synchronous invocation of the debit microservice and the synchronous placement of the credit request in the queue are performed within a classical distributed transaction. This transaction is ACID-based and is protected by a two-phase-commit protocol, as discussed in section 2.2. The purpose of this transaction is to ensure that Account_1 is never reduced without sending the request to increase Account_2. Similarly, retrieving the credit request message from the input queue of the credit microservice and increasing Account_2 through a database manipulation is also carried out within this distributed transaction. The application practically implements a funds transfer through these two classical distributed transactions and the guaranteed transfer of the message between the banks.

Microservice-based applications and cloud environments use the technique of ensuring *message integrity* to combine getting a message from or putting a message into a queue with associated data manipulations using a distributed transaction. This technique guarantees that messages requesting data manipulations and associated data manipulations themselves are always processed atomically, thus maintaining the integrity of in-flight information represented by messages and associated data

in data stores. Since only two resource managers are involved in such a distributed transaction, it is a practical approach for microservice-based applications and cloud environments.

3.4 Concurrency Control

Having multiple instances of the same microservice is necessary to ensure scalability, elasticity, and availability. These instances will work together to process requests that come in. Multiple instances of the same microservice may compete for requests on the same message queue, which is known as competing consumers [18]. However, this can cause problems with concurrency in certain situations, such as lost updates or inconsistent reads [7]. For instance, if the data used by the microservices do not have their own concurrency control features (like files or HTTP resources), two instances of the microservice might try to update the same data simultaneously.

To avoid concurrency problems, the application controls concurrent access to the data. Standard techniques like locking or timestamp ordering [7] are used to manage concurrent access. Locking is more commonly used in practice [19], and it comes in two forms: pessimistic locking and optimistic locking. Pessimistic locking is employed when conflicts are expected to occur frequently, while optimistic locking is used when conflicts are expected to be rare.

Pessimistic locking involves an application acquiring a lock on a data item before accessing it. The lock obtained must correspond to the access intent of the application. If a compatible lock already exists for the data item, such as an existing read lock that matches a requested read lock, access is allowed. However, if an incompatible lock exists, such as an existing read lock that does not match a requested update lock, access is denied. The application checks existing locks, verifies their compatibility, and takes appropriate action based on the result. On the other hand, optimistic locking does not acquire locks before manipulating a data item. Instead, it checks for conflicts without locking. A client can use conditional requests when updating an HTTP resource [20], for example. The client compares the values of the conditions, such as the entity tag or last modified timestamp, which were either cached or obtained through a prior request.

4. Compensation-Based Transactions

To support transactions in cloud-based applications, developers often have to implement the corresponding unit of work within the application itself. A microservice's updates may be erased by conflict resolution (as discussed in section 3.2), or lock conflicts (as discussed in section 3.4) may cause the rolling back of individual microservices that belong to a unit of work. Therefore, the unit of work concept is endangered because functions that are grouped together may experience failures that are detected only after other functions have been completed successfully. Moreover, a

unit of work's steps may manipulate data that the resource manager does not safeguard, i.e., their manipulations may not be simply requested to be undone. Developers can establish units of work by using compensation-based transactions in such scenarios.

4.1 Fundamental Principle Behind Rollback

In order to undo updates that have already been applied to data items, when a transaction needs to be rolled back, classical database systems [6] use persistent log records that store the values of the modified data items before they were updated. During rollback, a process called (transactional) *recovery* the database system simply resources the "pre-images" of modified data items. To avoid concurrency conflicts (see section 3.4), the database system holds locks on behalf of a transaction on the modified data items. The database system holds locks on modified data items on behalf of a transaction, preventing other transactions from accessing the data until the end of the transaction is reached. This ensures that other transactions will never see "dirty updates", i.e., intermediate updates that will later be undone, and will not make any decisions based on such data.

The rollback of a transaction ensures that the world is restored to its previous state, as if the transaction never happened. In classical transactions, the effects of the rolled back transaction are completely undone, representing a fundamental principle. However, compensation-based transactions violate this principle (as discussed in section 4.3), where the effects of a transaction cannot be completely undone, and some effects may remain even after the transaction is rolled back.

4.2 Dawn of Compensation

"Long-running" transactions can negatively impact the system's performance and scalability by decreasing concurrency and increased latency. To avoid this, transactions should be designed to complete their work as quickly as possible and release any locks they hold. We can also break down the work into smaller transactions or sub-transactions, each of which completes quickly and releases its locks. This reduces the duration of the locks and increases concurrency.

To help address long-running transactions, experts recommend several approaches. Some approaches do not use locks [7] and are usually made by a database system. Others are documented as patterns [19] for applications to use. For example, if an app wants to update a significant amount of data items at once, it can split them up into smaller sets called "mini-batches", that can be processed in a single transaction. Each mini-batch transaction with a counter to keep track of the last updated item. The following transaction reads that counter and continues with the mini-batch. This approach is called a "chained transaction" scheme.

If one of the mini-batches in this "chained transaction"cannot be processed, atomicity is broken, and all previous mini-batch updates must be undone. But as these updates have already been committed,

they cannot be automatically undone by restoring pre-images. To address this, the application must use *compensation logic* [21] to reverse previous updates. This logic is implemented as another chained transaction.

4.3 Semantic Recovery

Transactional recovery, as discussed in section 4.1, involves restoring pre-images and is a "mechanical" process that is unaware of any application-specific logic. Conversely, *semantic recovery* applies compensation logic specific to the application to rollback a transaction. This is because it is necessary to understand the semantics of the application to undo a failed transaction. Rolling back a chained transaction is an example of semantic recovery. Transactions that rely on compensation logic are called *compensation-based transactions*. Note that application programmers do not need to take special actions to enable the restoration of pre-images during transactional recovery, which can be performed uniformly across all applications. In contrast, they must explicitly provide programs that implement compensation logic for semantic recovery. Therefore, transactions that use semantic recovery are called compensation-based transactions.

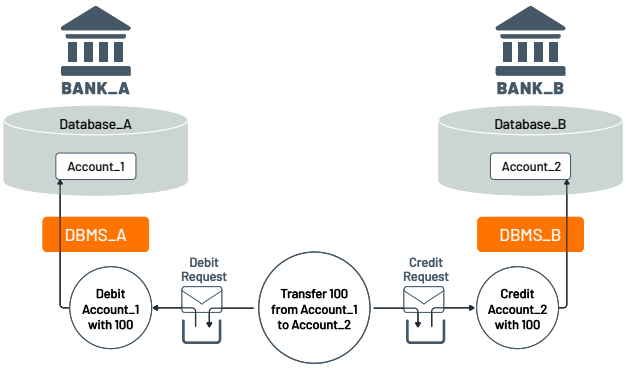

Figure 9: Funds transfer based on message-queuing

Expanding on our funds transfer example, we will illustrate how semantic recovery is utilized in such scenarios as shown by Figure 9. Typically, funds transfers occur asynchronously by a third-party using a transfer microservice. The transfer microservice starts a distributed transaction by placing a debit request message into the input queue of Bank_A's debit microservice and a credit request message into the input queue of Bank_B's credit microservice. The transaction is then committed. If placing a request message into either queue fails, the other message is removed from its queue, and the transaction is rolled back. In this way, neither Account_1 nor Account_2 are affected, and consistency is preserved. Assuming the transaction is committed successfully, the debit and credit microservices will later receive their corresponding request messages from their input queues and update the targeted accounts. Both microservices will signal their success to the transfer microservice (not shown in Figure 9), resulting in a successful funds transfer.

If the debit request fails because Account_1 would exceed its credit limit, the credit microservice will still process the credit request and increase the balance of Account_2, which violates consistency. To fix this, the increase of Account_2 must be undone. The debit microservice will send a fault message to the transfer microservice indicating the failure of the debit request. The transfer microservice will initiate another transaction and send a new debit request to the debit microservice of Bank_B to decrease the balance of Account_2. This will reestablish consistency between the balances of both accounts. It is important to note that consuming the fault message and submitting the new debit request are separate transactions.

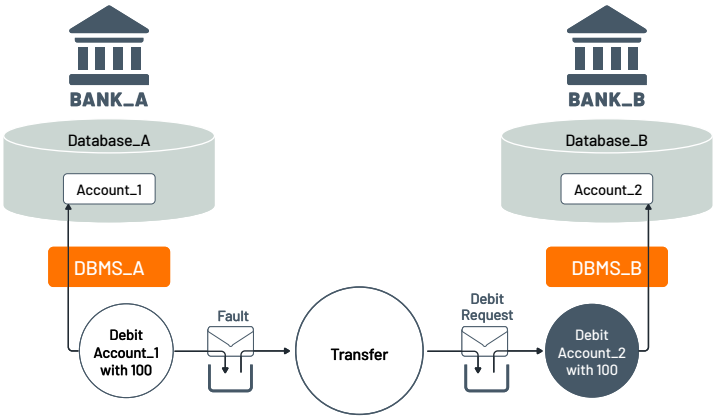

Figure 10: Initiating a compensation action

Figure 10 shows that semantic recovery appears to follow the main principle of recovery since it did not debit Account_1 or credit Account_2, leaving the world in the same state as before the transaction. However, inconsistencies may occur in the system if Account_2 was decreased by another transaction in the time between the temporary credit and the eventual debit. Therefore, semantic recovery may have limitations in ensuring complete consistency in all scenarios.

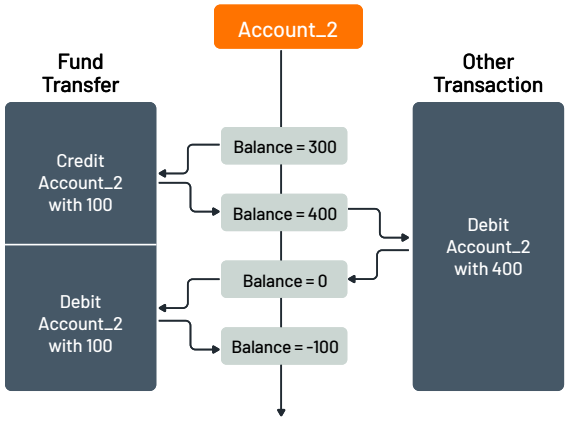

Figure 11: Sample of a potential side-effect of semantic recovery

Figure 11 shows an example of the limitations of semantic recovery. Assume Account_2 had 300 units before the credit microservice added 100 units, resulting in a balance of 400 units. Then, another

transaction reduced the balance by 400 units, leaving Account_2 with 0 units. Finally, the debit microservice tries to decrease the account by 100 units to compensate for the failed debit, but the balance becomes negative (-100 units). As a result, an overdraw charge is incurred, and the account holder must pay it.

When implementing semantic recovery in practice, it is essential to consider the limitation that compensation-based transactions, such as semantic recovery, generally do not uphold the fundamental principle of recovery (as discussed in section 4.1).

It is also important to consider that if the compensation action on Account_2 fails, such as when the debit microservice fails because Bank_B's policy does not allow overdrawn accounts, the overall consistency of the transaction is completely broken. This requires more complicated actions to be taken, and even human intervention may be necessary. To simplify this process, practitioners commonly use that the compensation action will not fail. While this may seem unrealistic, it is similar to the assumption made in database technology where complex processing must be performed if recovery fails. Therefore, it is essential to consider the possibility of compensation failure and to have contingency plans in place to address such situations. Workflow technology, as discussed in section 4.4, can be useful in such scenarios.

Many applications use compensation-based transactions. Pairs of functions exist in a domain where one function compensates the other, such as debit and credit, book and cancel, put and get, block and release, etc. and are referred to as *compensation pairs* [22]. Implementing these pairs does not require extra implementation efforts as they are very common in applications. Middleware, such as workflow systems, supports these transactions and can even perform semantic recovery automatically (see next section).

4.4 Compensation Spheres

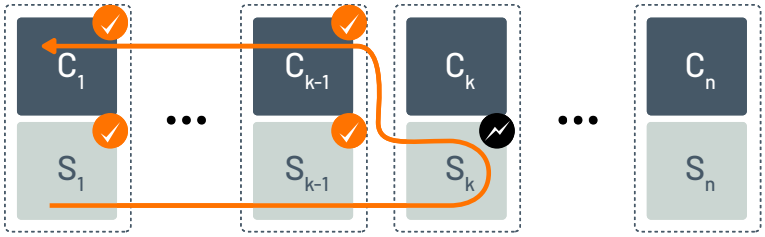

Figure 12: Rollback in SAGAs

Figure 12 shows one of the first systematic uses of compensation pairs, the SAGAs [23] concept. A SAGA is a collection of compensation pairs (Si,Ci) where each step (Si) and its associated compensation function (Ci) are implemented as a classical transaction. When a SAGA is executed, it runs the steps S1,S2,...,Sn in sequence. If no error occurs, the SAGA finishes successfully. In case of

an error, for example, if step Sk fails (and is, thus, rolled back), the already successfully finished steps S1,S2,...,Sk-1 are undone by invoking their compensation functions in reverse order, i.e., Ck-1,...,C1 will be performed. This means that the SAGA is rolled back and can be retried. Note that SAGAs inherit the problems sketched above, i.e., they violate the fundamental principle of rollback and assume that compensation does not fail.

In practice, applications require more flexibility than the strict sequence of executing steps as seen in the SAGA model. Control flow structures that a typical programming language supports like branches, loops, etc. are required, and parallel execution is also needed. For example, when a microservice sends messages to other microservices that can process them independently, as in our earlier scenario. In this case, some steps may not require a compensation action because there is no need to undo them, such as S1 in Figure 12. Some other steps may be grouped together with the semantics that if one of the steps fail, all completed steps before it will be compensated. This group is referred to as a *compensation sphere* [24]. By default, when a compensation sphere is rolled back, the control flow graph included in the sphere will be reversed and instead of the steps, their compensation actions will be executed when following the reversed control flow graph. Note that this is an immediate generalization of SAGAs. In addition, a sphere, like S in Figure 12, may have its own compensation action associated with it, like C in Figure 12. When an error occurs within the sphere, it may be compensated by running the compensation action attached with the sphere itself, called *integral compensation* [24], or by running the compensation actions of the steps contained within the sphere, called *discrete compensation* [24]. It may even be decided that a compensation sphere is rolled back after the control flow leaves the sphere, like in case an error is detected at a later point in time. These control flow structures are called *microflows* - these will be discussed in a separate paper.

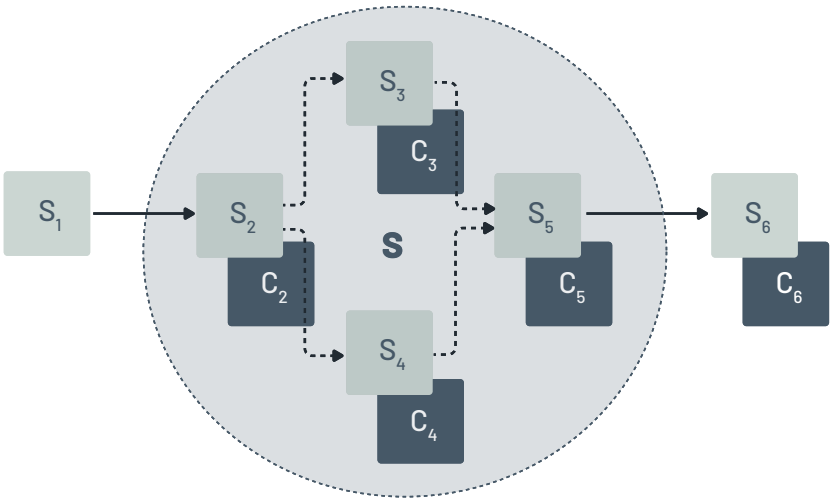

Figure 13: Compensation spheres in workflows

Figure 13 highlights that microflows are a type of *business process* known as *workflows*, which can involve human interaction and interruptions [22]. For instance, if a compensation action fails,

humans can be notified to correct the error and the workflow can resume. Workflow languages, such as BPEL [26] or BPMN [27], make minor adjustments to support compensation spheres. A graphical representation of BPMN, similar to Figure 13, is shown in Figure 14.

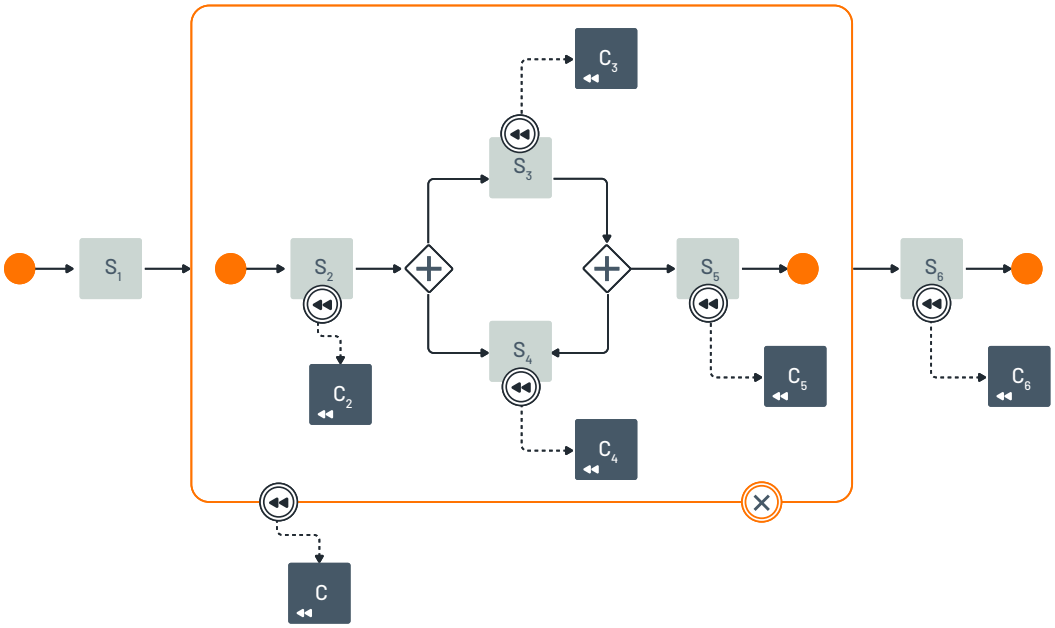

Figure 14: Compensation sphere in BPMN

Summary

In this white paper, we discussed the ACID paradigm, which is the core concept of classical transactions, and how it relates to the idea of a sphere of control. We then outlined the basic issue with transactions that use multiple resource managers, specifically atomic commitment, and how the two-phase-commit protocol solves this problem. We also examined data replication and its impact on the transaction concept. The rise of highly distributed applications, prompted by cloud native architecture and microservices, was also reviewed. This has resulted in the need for new consistency models that rely on the BASE paradigm, which reflects the CAP theorem. As a result, the ACID properties must be sacrificed, particularly in cloud native and microservice-based applications. To enable a suitable unit of work concept that supports proper recovery, we analyzed the fundamental principle of rollback. We introduced semantic recovery as a generalization appropriate for highly distributed and long-running applications. We also discussed compensation pairs as the basis of semantic recovery, as well as partial rollback of applications based on compensation spheres. All of these concepts together established the notion of compensation-based transactions.

In summary, the key findings regarding transactions in a microservice environment are given below.

- Using classical, ACID-based transactions for individual microservices is an appropriate approach.
- It is feasible and practical to involve two resource managers for microservice-based applications in a distributed transaction that is protected by the two-phase-commit protocol.
- Microservice-based applications, particularly in the cloud, must handle soft-state effectively by carefully considering consistency and availability.
- It is necessary to implement the functionality of relevant microservices as compensation pairs if several microservices need to be grouped into a unit of work.
- The compensation-based transaction concept is suitable for microservice-based applications.

References

1. Atomikos, 2023, https://www.atomikos.com/Main/WebHome.
2. Bernstein, Philip A., and Eric Newcomer. Principles of Transaction Processing, 2nd ed., Morgan Kaufmann, 2009.
3. Bernstein, Philip, et al. Concurrency Control and Recovery in Database Systems, Addison-Wesley, 1987.
4. Brewer, Eric A. Towards Robust Distributed Systems: Principles of Distributed Computing, 2000.
5. Brewer, Eric A. "CAP Twelve Years Later: How the 'Rules' Have Changed." InfoQ, 2012, https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/.
6. Davis, C. T. "Data Processing Spheres of Control." IBM Systems Journal, vol. 17, no. 2, 1978, pp. 137-54.
7. Eswaran, K. P., et al. "The Notion of Consistency and Predicate Locks in Database Systems." Communications of the ACM, vol. 19, no. 11, 1976, pp. 624-33.
8. Fox, Armando, et al. "Cluster-Based Scalable Network Services." Proceedings of the Symposium on Operating Systems Principles, 1997, pp. 78-91.
9. Fowler, Martin. Patterns of Enterprise Application Architecture, Addison-Wesley, 2003.
10. Garcia-Molina, Hector, and Kenneth Salem. "Sagas." Proceedings of the ACM SIGMOD, 1987, pp. 249-59.
11. Gilbert, Seth, and Nancy Lynch. "Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services." ACM SIGACT News, vol. 33, no. 2, 2002, pp. 51-59.
12. Gray, Jim. "The Transaction Concept: Virtues and Limitations." Proceedings of the International Conference on Very Large Databases, 1983, pp. 144-54.
13. Gray, Jim, and Andreas Reuter. Transaction Processing: Concepts and Techniques, Morgan Kaufmann, 1993.
14. Haerder, Theo, and Andreas Reuter. "Principles of Transaction-Oriented Database Recovery." ACM Computing Surveys, vol. 15, no. 4, 1983, pp. 287-317.
15. Hohpe, Gregor, and Bobby Woolf. Enterprise Integration Patterns, Addison-Wesley, 2004.
16. JBoss. Narayana, 2023, https://www.narayana.io.
17. Korth, Henry F., et al. "A Formal Approach to Recovery by Compensating Transactions." University

of Texas, 1990.

18. Leymann, Frank, and Dieter Roller. Production Workflow, Prentice Hall, 2000.
19. Leymann, Frank. “Supporting Business Transactions via Partial Backward Recovery in Workflow Management Systems.” Proceedings of the Conference on Datenbanksysteme in Büro, Technik und Wissenschaft, 1995, pp. 279-97.
20. Leymann, Frank. Microflows, WSO2, 2023, [to be added].
21. Leymann, Frank, et al. WSO2 REST API Design Guidelines, WSO2, 2016, https://wso2.com/whitepapers/wso2-rest-apis-design-guidelines/
22. Nygard, Michael T. Release It!, Pragmatic Bookshelf, 2007.
23. Pritchett, Dave. “BASE: An ACID Alternative.” ACM Queue, vol. 6, no. 3, 2008, pp. 48-55.
24. OASIS. “Web Services Business Process Execution Language Version 2.0.” OASIS, 2007, http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html.
25. Object Management Group. “Business Process Model and Notation (BPMN) Version 2.0.” Object Management Group, 2011, https://www.omg.org/spec/BPMN/2.0/PDF.
26. The Open Group. “Distributed Transaction Processing: The XA Specification.” The Open Group, 2017, https://publications.opengroup.org/c703.
27. Seata, 2023, https://seata.io/en-us/index.html.