Ensuring Data Consistency in Microservices Based Applications

Todd Little – Chief Architect, Transaction Processing Products

Executive Summary

Microservices architecture is increasingly being adopted by organizations to improve scalability, isolation, fault tolerance, faster time to market, and more. Because of these advantages over traditional monolithic application development, many monolithic applications are being decomposed into a set of microservices. However, developing applications in a microservices world provides some challenges related to data consistency. In traditional monolithic application development, typically a single data source is used and accessed by a single application. The only transactional requirements are for local transactions. At most a couple of data sources are used such as a database and a messaging system and the application platform or application server provides transactional consistency across them. In the microservices world, multiple data sources are involved with each microservices typically having its own data source(s). Now a typical transaction spans multiple microservices and multiple data sources. This presents a consistency problem without some sort of distributed transaction coordination.

The following example describes the issue with data consistency. Let's consider that you are transferring money from one financial institution to another financial institution by using the microservices that each institution provides. Typically, this is done with an actor such as a bank teller that withdraws money from one account and deposits it in another. This might look like the following set of state transitions for transferring 100 from account A to account B:

State	Action	Teller "balance"	Account A balance	Account B balance
S0		0	500	700
S1	Withdraw 100	100	400	700
S2	Deposit 100	0	400	800

If all goes well, both the withdraw action and the deposit action both occur. Presumably if the Withdraw fails, the teller won't even try the Deposit, so the state remains S0. However, if the Withdraw succeeds and the Deposit fails, then the teller is left with 100 from Account A. To help alleviate this problem, applications can rely upon the services of a transaction coordinator to ensure either both the Withdraw and Deposit requests succeed or they both fail.

Introduction to Distributed Transactions

Distributed transactions are those transactions that typically span multiple sources of data or multiple participants accessing the same data. Traditional monolithic applications use local transactions, meaning transactions that are started and completed by a single participant and only interact with a single data source. Once an application becomes distributed or accesses multiple data sources some

coordination is needed in the form of a distributed transaction. Distributed transactions are used in two cases:

1. Transactions that span multiple data sources
2. Transactions that span multiple application components that access a single data source

A common example of a distributed transaction is something that transfers money from one microservice to another microservice. The issue is that if part of the transfer is completed, say the withdrawal, and the other part of the transaction, say a deposit fails, at the end of the transaction, the money transferred is in limbo. It's no longer in the account the money was withdrawn from, and it's not in the account that was supposed to receive the transfer. This results in an inconsistent state between the two microservices.

To alleviate this problem a transaction manager (also called a transaction coordinator) is used to ensure that either both the withdrawal and the deposit occur successfully, or both fail. This is the most common model for a distributed transaction. There are other models that allow for one part to complete and the other part to fail which then requires the first part to be compensated or rolled back. These are called eventual consistency models as there are points of time when the data among the participants isn't consistent.

ACID – Atomicity, Consistency, Isolation, Durability

The ideal solution for distributed transactions is one that provides the ACID properties typically associated with transactions.

Atomicity

Atomicity is the property of a transaction such that all changes to all participants either succeed or fail.

Consistency

Consistency or sometimes called correctness is a property that the participants move from one valid state to another. This means there are no externally visible changes to one participant that are inconsistent with the changes to another participant. The participating resources (databases, queuing systems, etc.) external view is always consistent.

Isolation

Isolation ensures that the participants transactions execute as though they were executed serially.

Durability

Durability ensures that once the outcome of a transaction has been decided, the decision is written to a persistent storage mechanism to allow recovery in the presence of failures.

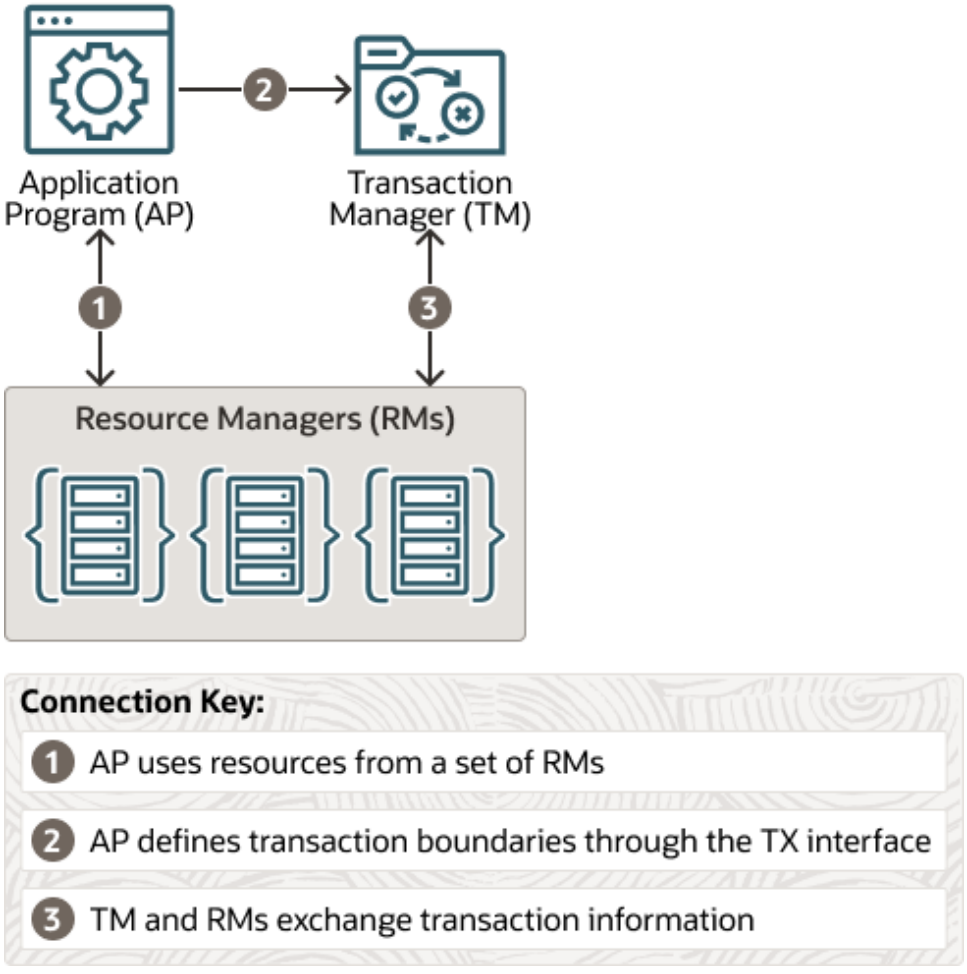
Common Transaction Models

There are several distributed transaction models to solve these consistency issues. The most widely known of these models is the XA Distributed Transaction Model. This model guarantees the ACID properties of a distributed transaction. It is widely implemented by application servers and transaction processing monitors. Another model is the Eclipse MicroProfile Long Running Actions or LRAs. LRAs are specific form of Sagas defined by a set of Java annotations. Sagas provide a form of eventual

consistency meaning that until the Saga has completed, each of the systems involved will likely be in inconsistent states with respect to each other and that are visible to others. The third model covered in this paper is the Try-Confirm/Cancel model. This model relies on resources being able to be placed in a reserved state. Once all the reservations are made, all the reservations are either confirmed or canceled placing the reserved resource back into an available status.

XA

The XA transaction model defines three components. The application (AP) which is the component that provides the business logic and demarcates transaction boundaries. Resource managers (RM) are the components that manage stateful resources such as databases, queuing or messaging systems, caches, and others. And finally, it defines a transaction manager (TM) that provides the transaction coordination amongst the participants.



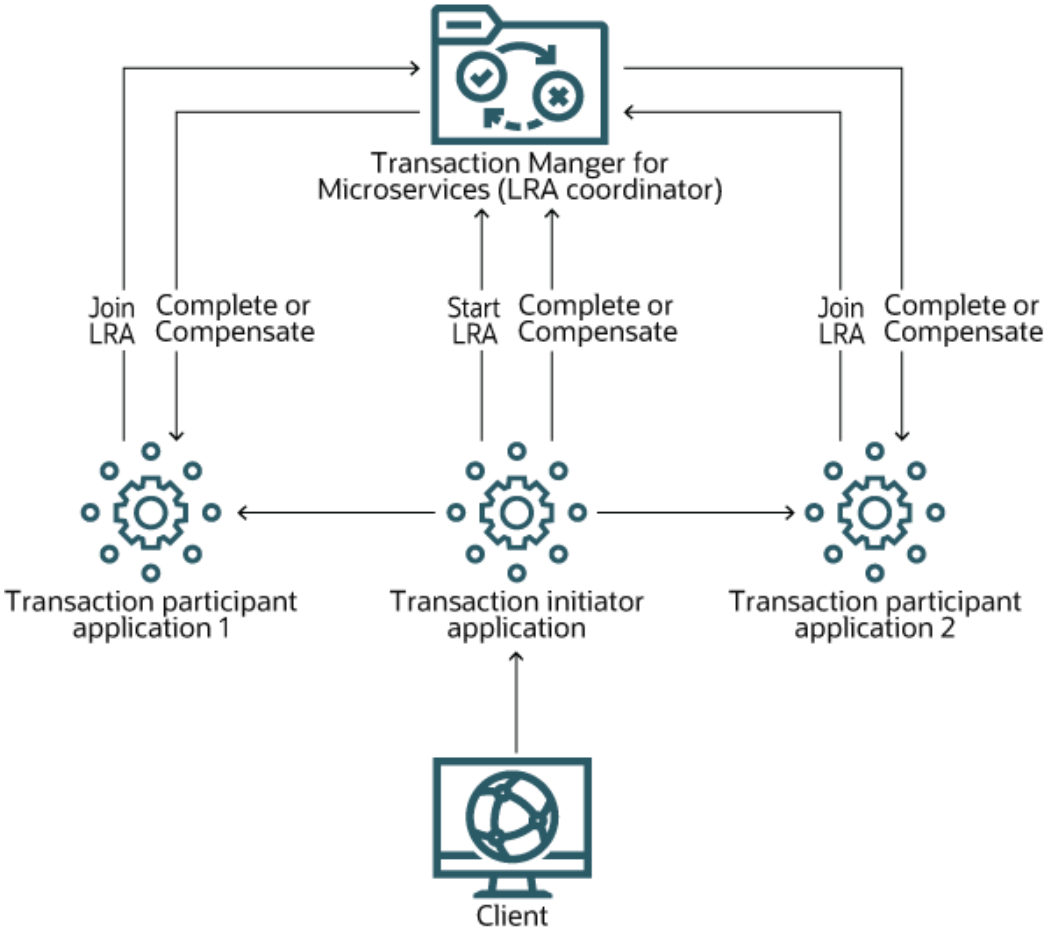
The basic flow is:

1. AP asks the TM to begin a transaction
2. AP updates one or more RMs using the transaction context
3. AP asks the TM to commit or rollback the transaction
4. TM asks all RMs to prepare or rollback
5. If commit, TM asks all RMs to commit

Sagas or Long Running Actions

In the Sagas or Long Running Actions (LRA) transaction model, participants enlist their involvement in the LRA by calling an LRA coordinator and providing it with callback URIs. The LRA coordinator uses these URIs to manage the transaction flow. Each participant uses local transactions that are independent from each other. The basic flow is:

1. The initiator of the LRA calls the LRA coordinator to begin the LRA and pass it callback URIs
2. The initiator calls one or more other participants passing along the ID of the LRA in headers
3. The other participants enlist in the LRA by calling the LRA coordinator passing their callback URIs
4. The initiator calls the LRA coordinator to either complete or compensate the LRA
5. The LRA coordinator calls each participant's complete callback URI or compensate callback URI depending upon whether the initiator asked to complete or compensate the LRA



Because Sagas use local transactions there will be periods when the overall state of the system is inconsistent with a goal of eventually being consistent at the end of the Saga. This is because the local transactions complete or compensate independently. As a result, there will be a period when one or more local transactions will have completed or compensated while others have not. Due to the lack of

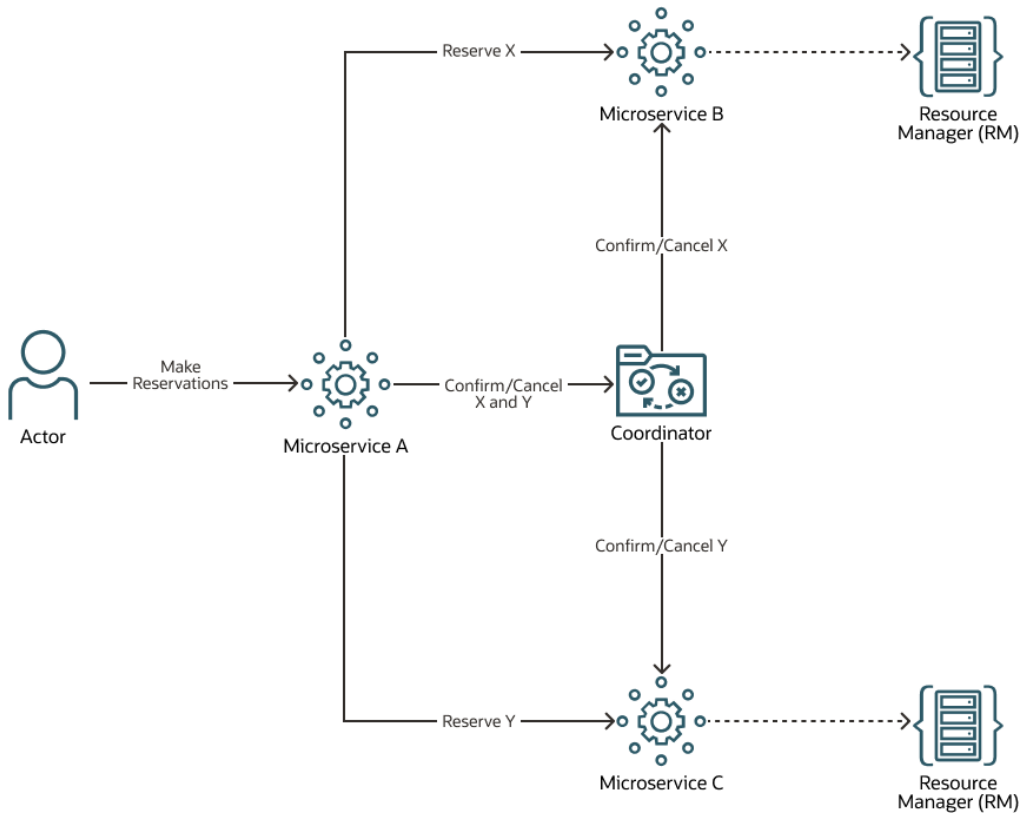
locking and isolation other systems or users will be able to see these inconsistent states and potentially make incorrect decisions based upon those inconsistent states.

Try-Confirm/Cancel

The Try-Confirm/Cancel (TCC) transaction model relies upon the participants in the transaction maintaining reservations, i.e., some resources that are held in a reserved state until either confirmed or canceled. The model relies on the basic HTTP verbs POST, PUT, and DELETE. A POST is used to create a new reservation. PUT is used to confirm the reservation while DELETE is used to cancel the reservation.

A transaction coordinator (TC) is often used to ensure that all participants either confirm their reservations or cancel their reservations. The basic flow is:

1. Initiator calls the TC to begin the TCC transaction
2. Initiator invokes POST (Try) on one or more other participants to create reservations
3. The participants will call the TC to indicate their participation in the TCC transaction
4. Initiator calls the TC to either Confirm or Cancel the TCC transaction
5. The TC then calls PUT (Confirm) or DELETE (Cancel) on each participant depending upon what decision the initiator made on the outcome of the transaction



Heuristic Outcomes

With any distributed transaction model there is always a possibility that the transaction will end up with an inconsistent outcome. An example of this in XA is that after all RMs have been prepared, one or more of them fails to commit. While this should be a highly unusual circumstance, it can happen. In eventually consistent transaction models such as LRAs, a participant that is asked to complete or

compensate may be unable to do so because of failures or due to intervening transactions leading to a heuristic outcome.

Timeouts are one of the more common reasons for a heuristic outcome as most transaction models provide a timeout mechanism to ensure progress is made. In the TCC model, each participant provides a timeout value indicating how long it will hold a reservation. After that period of time, the participant can unilaterally decide to cancel the reservation. This might happen even as the transaction is being confirmed which can lead to some participants confirming the transaction and some canceling the transaction. Unfortunately, in many cases heuristic outcomes require some sort of manual intervention to resolve and should be avoided as much as possible.

Advantages and Disadvantages of Each Model

It's likely that one or more of the above transaction models could be used in any given application depending upon the consistency and performance requirements. In this section we'll cover the advantages and disadvantages of each transaction model with respect to consistency, performance, and required developer effort.

XA

While the XA transaction model provides the strongest consistency guarantees, it is often considered an anti-pattern in the world of microservices. The primary reasons for XA being considered an anti-pattern is that it adds additional coupling between microservices and can lead to significant performance impacts. The claim about additional coupling is largely a red herring as presumably there is some business requirements (coupling) that the microservices must adhere to for the sake of consistency. The claim about performance is real, although proper application design can make the performance impact over other transaction models far less significant.

The major benefits of XA transactions are:

1. Global ACID properties – all the participants move from one consistent state to another, with complete isolation and serializability
2. The programming model places minimal requirements on the application, with the application only determining the boundaries and desired outcome of a transaction without any need to code compensating actions.

However, there are potential performance issues when using XA transactions, and it is largely related to the necessary locking the RMs must perform to ensure serializability. RMs lock the resources that have been read, written, or deleted while the transaction is in process. This means that other transactions using those same resources must wait until those locks are released. This serialization of requests waiting for these locks can significantly limit the performance of an application and possibly introduce deadlocks.

Significant performance problems using XA are typically caused by the design of the application. Applications that make updates to an RM should avoid making updates to common resources. An example might be maintaining a balance value that tracks the sum of a set of records. If that balance is updated every time one of the records is updated, it will force the serialization of the updates to those records as each update will have to wait for the lock on the balance. The problem becomes worse as the length of a transaction increases. If these transactions take 2 seconds to complete, the maximum

throughput that can be achieved would be 30 transactions per minute. Not a scalable solution and the major reason for XA being considered an anti-pattern for microservices.

If maintaining or obtaining the balance of the records was done by a database query instead of updating some balance record on each request, then there would be no serialization of the updates to those records. Using the above same 2 second time to complete a transaction, the throughput is essentially unlimited as an unlimited number of requests could be handled in parallel.

Other examples of poor application design include:

1. Having a counter that is updated by each request. Assuming that counter is shared across requests, will cause all requests to be serialized.
2. Any access to a resource that is shared by many or most transactions

A properly designed application that minimizes lock conflicts can perform nearly as well as an application that doesn't ensure any consistency. As mentioned, the major impact will be on latency as there are more network requests involved in coordinating the transaction than if the application didn't ensure consistency.

Sagas or Long Running Actions

Sagas or Long Running Actions (LRAs) were designed to handle situations where using XA might not be feasible or appropriate. Because of the locking involved with XA transactions, it's generally recommended that XA transactions be relatively short lived involving only machine to machine interactions. Where Sagas or LRAs come into play is when there are users involved in the decision making for a transaction or for long workflows that may execute over minutes to hours or more. The major advantage of Sagas or LRAs is their locks are held only for the duration of the local transactions, not the entire Saga or LRA, so they may not introduce as many serialization performance issues.

While avoiding serialization performance issues is great for performance, Sagas or LRAs places some significant burdens on the application. Specifically, when a Saga or LRA needs to be aborted or canceled, application logic is required to perform the appropriate compensating action. This may sound easy as one can trivially compensate a deposit with a withdrawal. Yet if another intervening withdrawal has taken place, it is conceivable that there aren't enough funds to make the compensating withdrawal. In this case it is likely that the compensating action would fail leaving the transaction with a heuristic outcome. Many other cases exist where it may be extremely difficult or impractical to implement compensating actions. As well, these compensating actions are up to the developer to create and they may contain bugs like any other code and can be difficult to test under all failure scenarios.

Try-Confirm/Cancel

The Try-Confirm/Cancel (TCC) transaction model has the same global consistency guarantee that the XA transaction model provides, yet with limits on the type of application that can leverage the TCC transaction model. TCC only works with application resources that can be held in reserve. For example, and airline seat or hotel reservation. With each reservation, the system moves from one consistent state to another. The model is completely scalable as there are no imposed serialization constraints. Like XA TCC is easy for the developer to utilize as the developer just needs to demarcate the transaction boundaries and determine the outcome of the transaction. The workflow to ensure all participants

either confirm or cancel can be handled by the transaction coordinator, further minimizing the responsibilities placed on the application code.

Choose the Model That Fits

As with many things, there is no one size fits all approach to distributed transactions. Each model has its own advantages and disadvantages and there is no reason to select only one. For things like financial transactions between applications where no user interaction is involved in the transaction, XA offers the best consistency guarantees with the least amount of developer effort. TCC offers similar consistency and minimal developer effort if the application can use a reservation model in its transactions. Sagas or LRAs provide the most flexibility at the cost of developer complexity.

Conclusion

Ensuring consistency across disparate microservices can be difficult to achieve when required. The level of consistency needed is determined by the business requirements for the microservices involved. Some microservices may require very strong consistency, while others may only require eventual consistency. As with many things, there is no one size fits all approach to distributed transactions. Each model has its own advantages and disadvantages and there is no reason to select only one. For things like financial transactions between microservices where no user interaction is involved in the transaction, XA offers the best consistency guarantees with the least amount of developer effort. TCC offers similar consistency and minimal developer effort if the application can use a reservation model in its transactions. Sagas or LRAs provide the most flexibility at the cost of developer complexity. Choose the distributed transaction models that best suits your business requirements.