Oracle 微服务事务支持

-- ORACLE OSaga & MicroTx

公益讲座11：00准时开始，请大家先浏览云技术微信公众号技术文章。资料会在各群同步发布，已入群客户请勿重复入群！

20-20

数据库和云讲座群

甲骨文云技术公众号

Oracle在微服务事务方面的支持

Oracle SEHub
贺友胜(Hysun He)
2023年2月

议程

背景

分布式事务协议

- XA / LLR
- TCC
- Saga (LRA)

Oracle数据库Saga支持

Oracle 微服务事务中间件

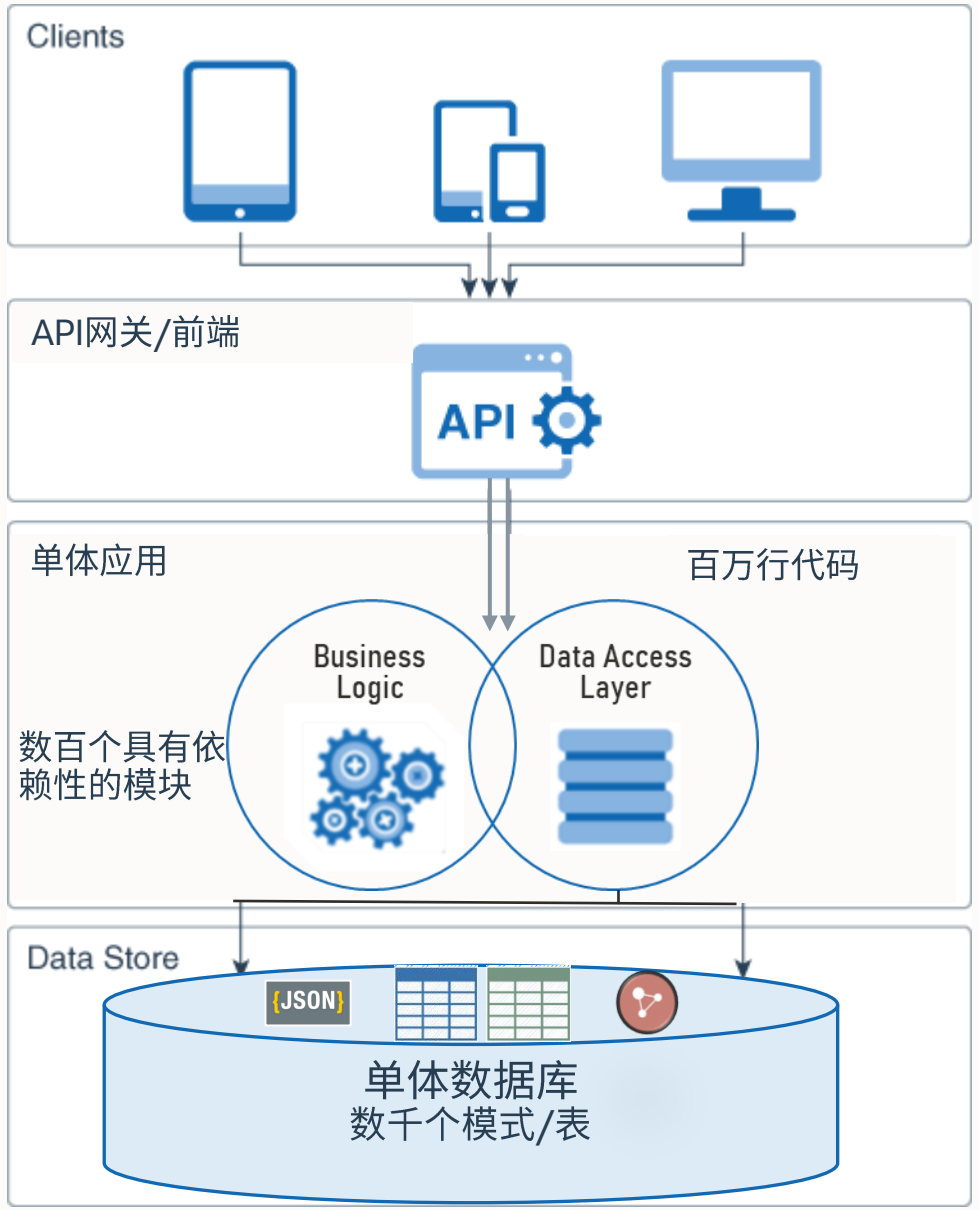
单体应用

优点

- 本地交易
- 单体数据库
- 集中式数据库管理员

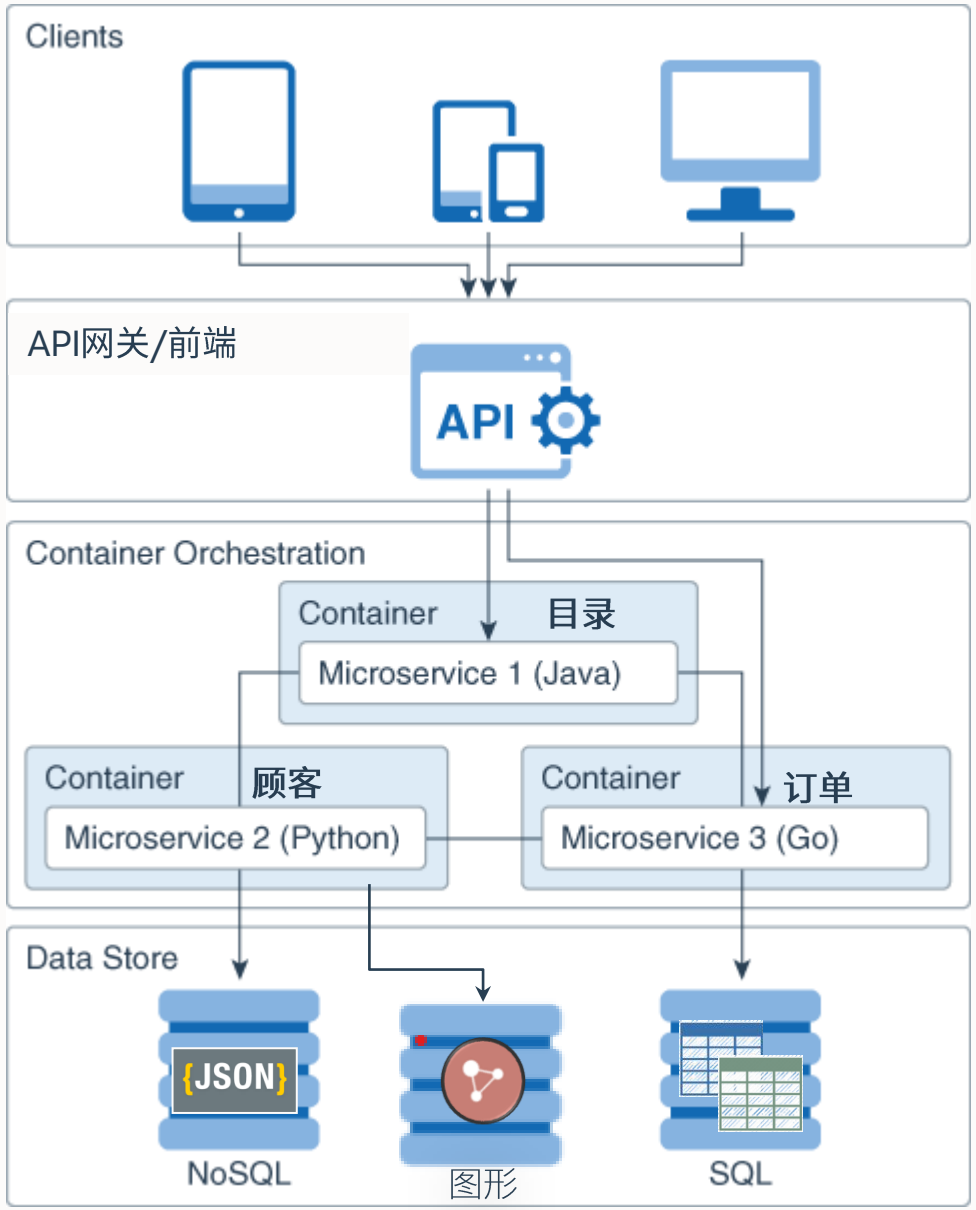
缺点/挑战

- （隐藏的）依赖森林
- 百万行代码
- 数百/数千张表
- 难以维护和调试
- 规模化成本高
- 发布周期长

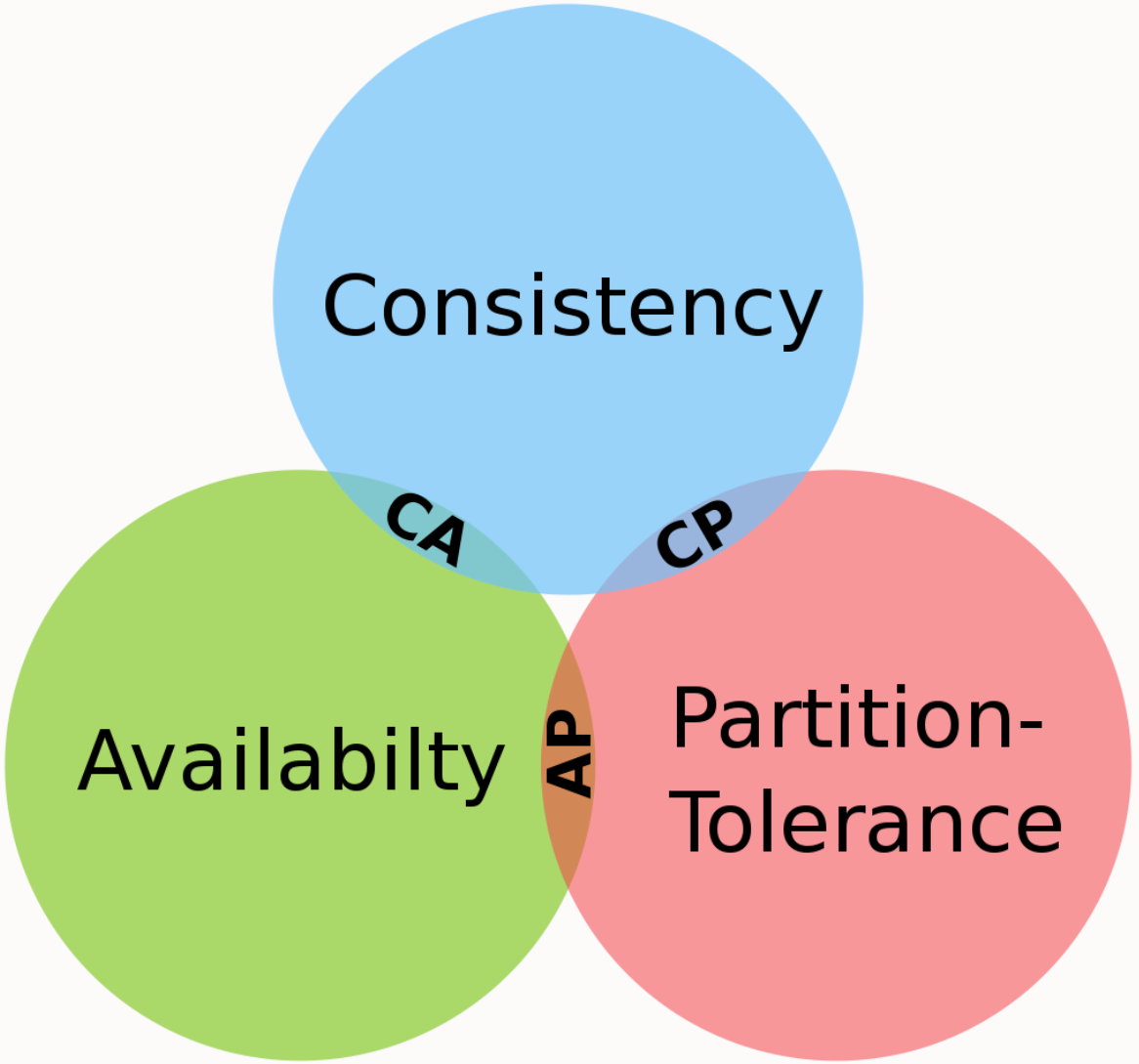


基于微服务的应用

- 优点
  - 独立单位
  - 易于开发/测试/部署
  - 多语言开发以实现更佳匹配
  - 易于维护
  - 在服务级别独立扩展
  - 更高的弹性/容错能力
  - 默认云架构模式
- 缺点/挑战
  - 编排复杂度
  - 管理多个数据存储
  - 更大的延迟



CAP理论回顾



Source: medium.com

议程

背景

分布式事务协议

- XA / LLR
- TCC
- Saga (LRA)

Oracle数据库Saga支持

Oracle 微服务事务中间件

分布式事务协议/模型

XA

a) 二阶段提交(2PC)
b) ACID
c) 全局事务(锁)；强一致性
d) 事务由RM控制

TCC

a) 二阶段提交(2PC)
b) ACID
c) 事务范围由业务层控制；强一致性

SAGA

a) 一阶段提交1PC
b) ACD
c) 本地事务(锁)；最终一致性

议程

背景

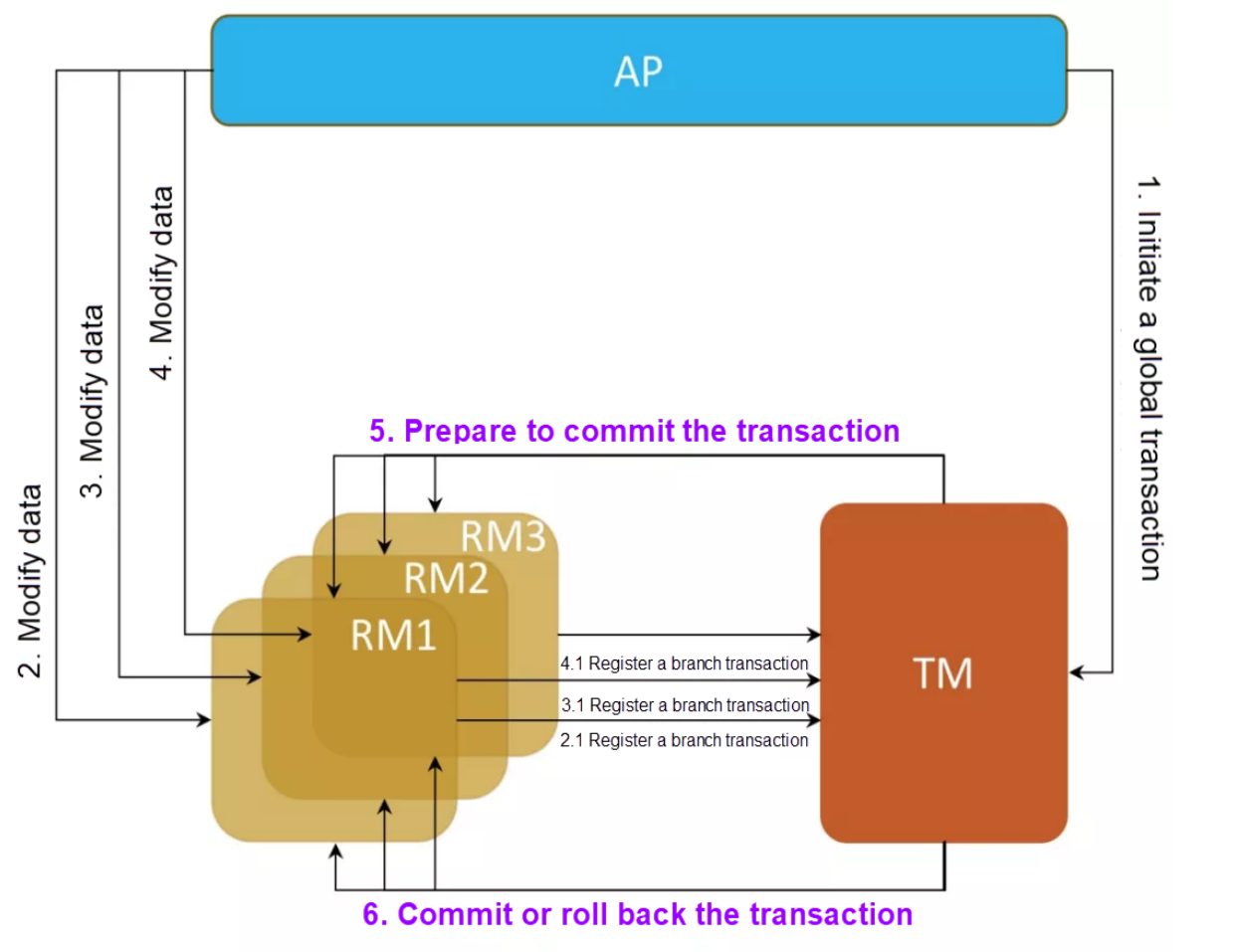
分布式事务协议

- XA / LLR
- TCC
- Saga (LRA)

Oracle数据库Saga支持

Oracle 微服务事务中间件

XA



特点

- 2PC
- 强一致性
- 全局ACID属性
- 使用更简单，几乎无额外开发工作量

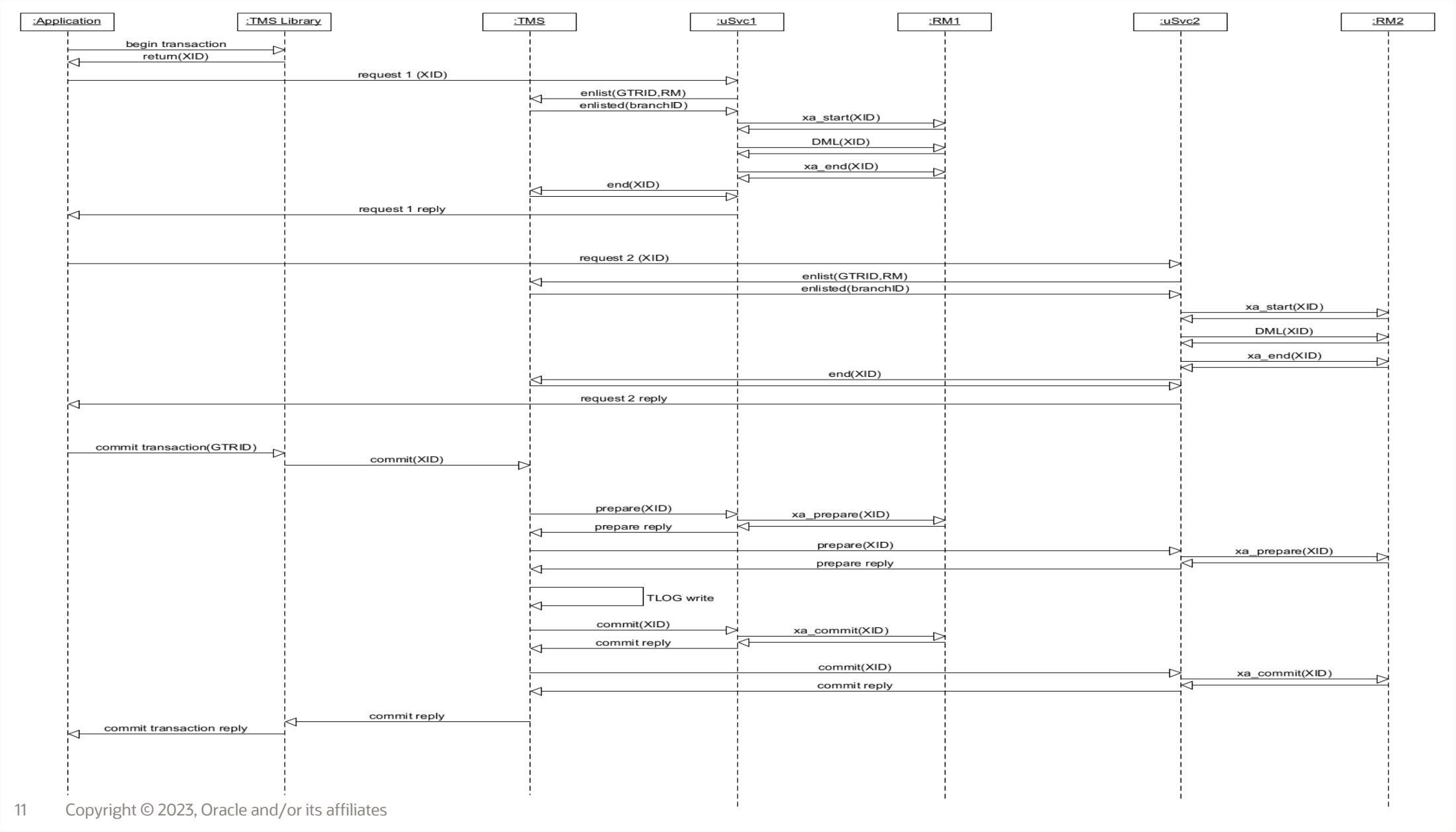
不足

- 性能：资源锁定时间长(等XA事务结束)，造成等待现象
- 容易引发程序串行化执行

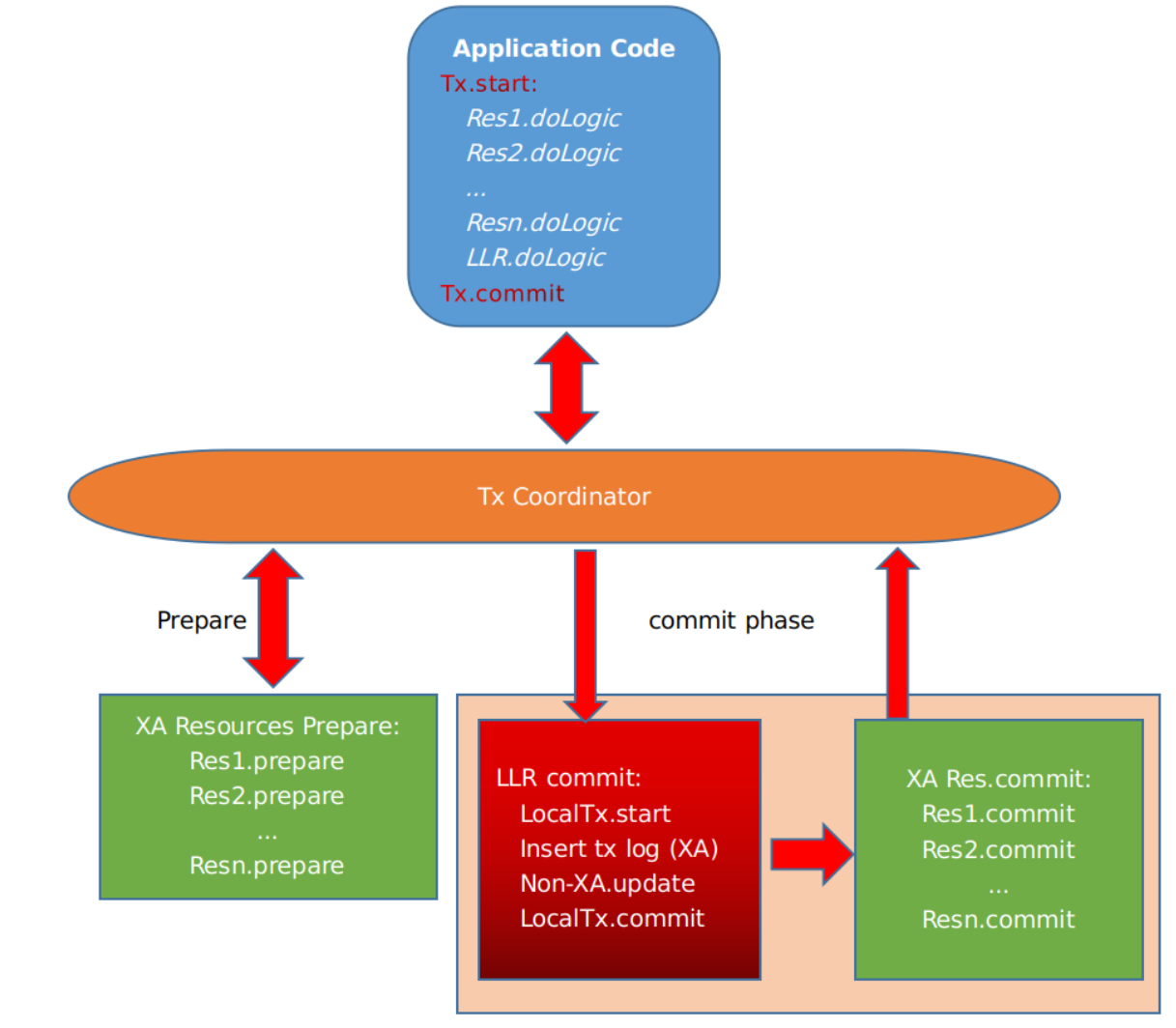
场景

- 需要强一致性保证的场景，如金融

XA - 2PC



XA扩展协议 - LLR (Logging Last Resource)



LLR

1) 允许一个 non-XA 数据源参与全局XA事务。
2) 不会破坏全局XA事务的ACID特性
3) 借助支持XA的数据源(如Oracle数据库表)来实现

执行原理

1) Prepare阶段:
   a) 在所有XA资源上先做prepare操作
   b) LLR不参与prepare阶段
2) Commit阶段：
   a) 首先处理LLR。在支持XA的关系数据库中建立一张事务日志表，先向日志表中插入一条日志记录
   b) 执行具体的此non-XA数据源的逻辑
   c) 提交LLR参与者的本地事务
   d) 提交其它支持XA的参与者的本地事务
   e) 提交整个XA事务

议程

背景

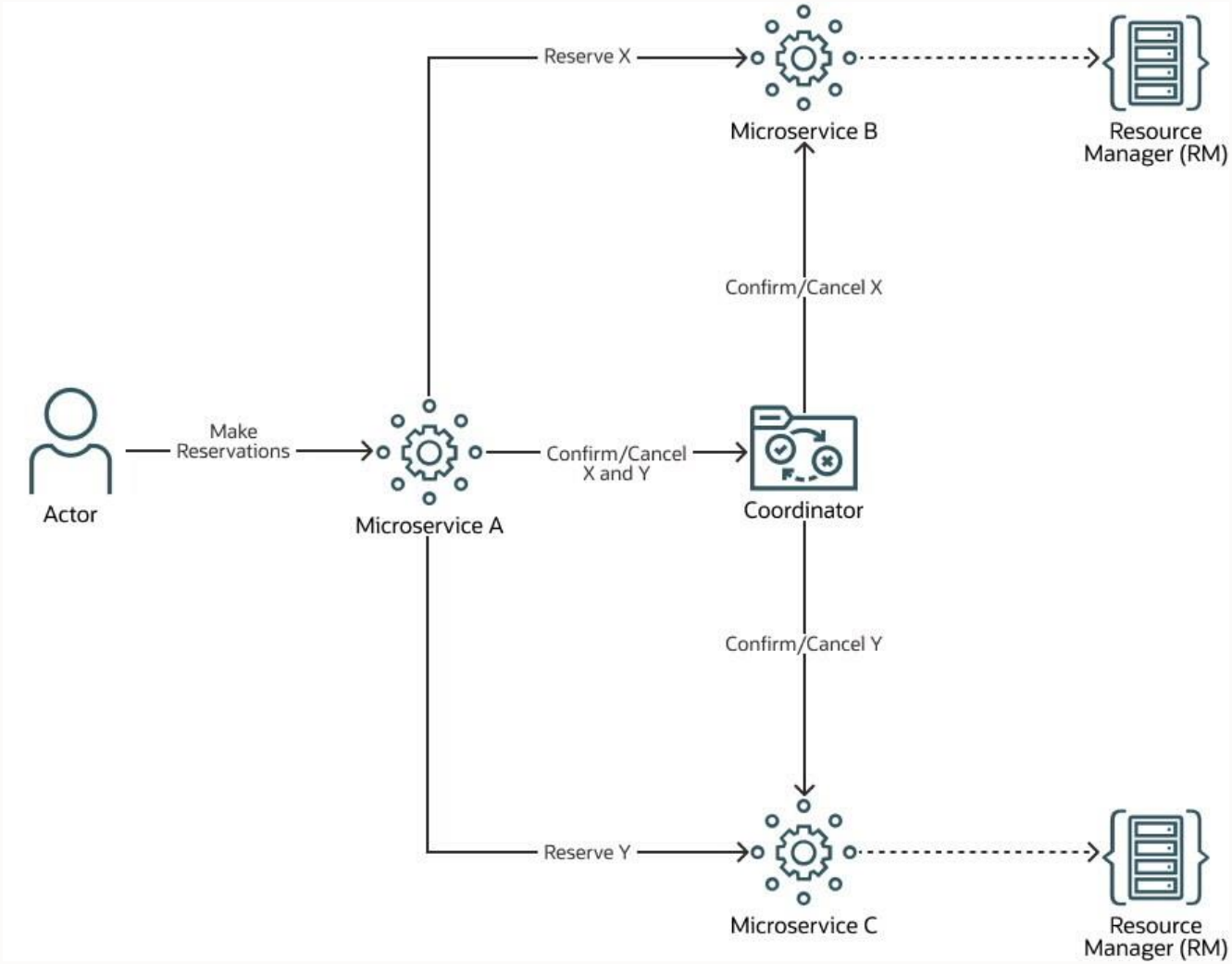
分布式事务协议

- XA / LLR
- TCC
- Saga (LRA)

Oracle数据库Saga支持

Oracle 微服务事务中间件

TCC事务模型 (Try...Confirm/Cancel)



特点

- 2PC
- 中间状态 (pending/reserved)
- 具有ACID特性
- 强一致性
- 补偿逻辑简单(相对LRA)

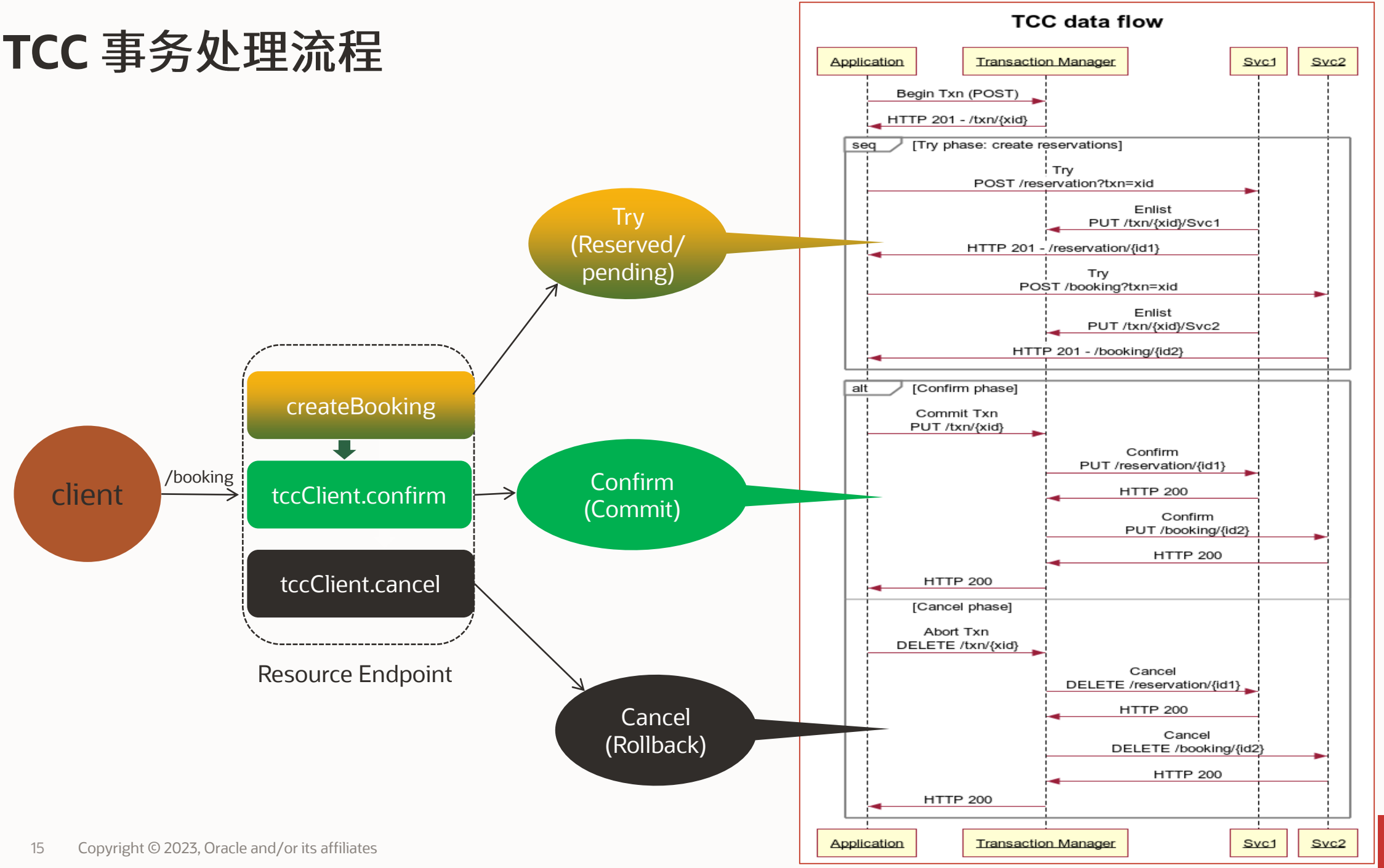
不足

- 需要设计中间状态
- 性能　XA < TCC < LRA(SAGA)

场景

- 资源预留(预订)模型。
- 强一致性要求

TCC 事务处理流程



议程

背景

分布式事务协议

- XA / LLR
- TCC
- Saga (LRA)

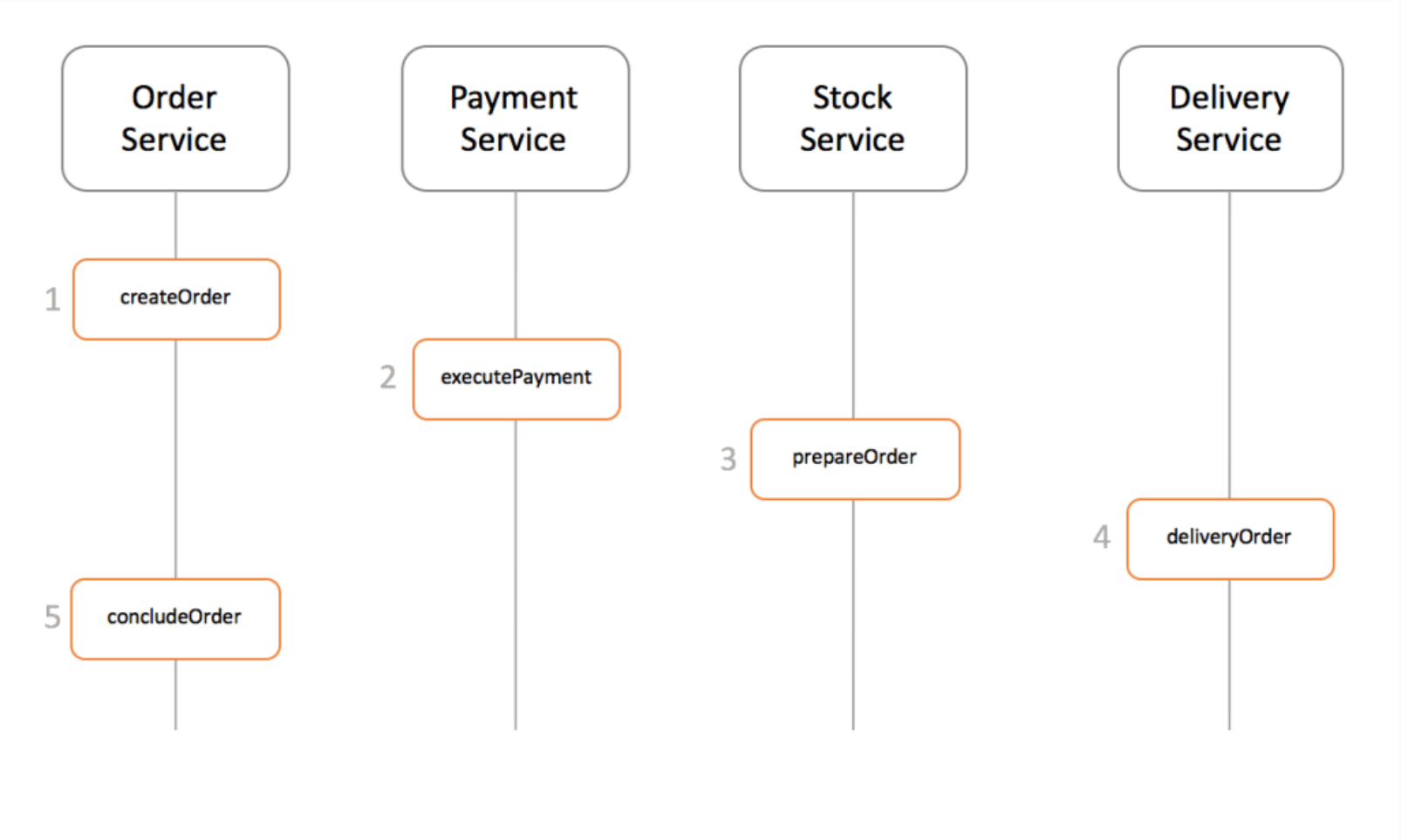
Oracle数据库Saga支持

Oracle 微服务事务中间件

SAGA 事务处理模型

SAGA是一种分布式事务处理模型

SAGA全局事务由一系列**本地**事务组成，各参与者只更新自己的本地数据。如果出现失败，则逆向补偿。



Source: Denis Rosa, Couchbase

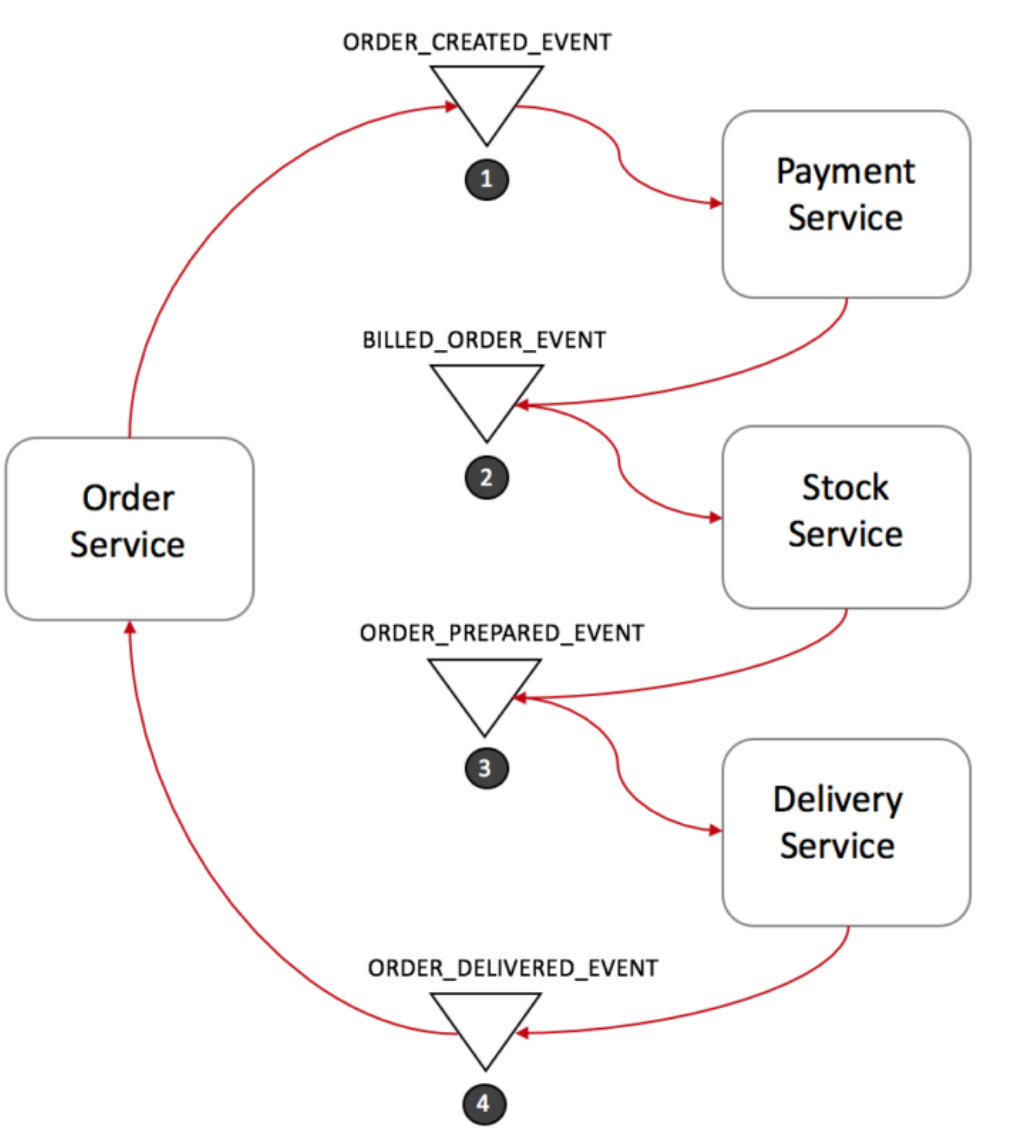
SAGA事务编排方式－Events/Choreography (无中央编排器)

说明

- 各事务参与者更新本地数据，如Order Service往自己库内插入一条订单记录
- 各事务参与者更新本地数据后，发布一条事件通知下一个参与者。如Order Service新建订单后，发布ORDER_CREATED_EVENT通知Payment Service
- 这种由上一个服务执行完后通过事件通知下一个服务的方式，叫 Events/Choreography方式，没有中央编排器。

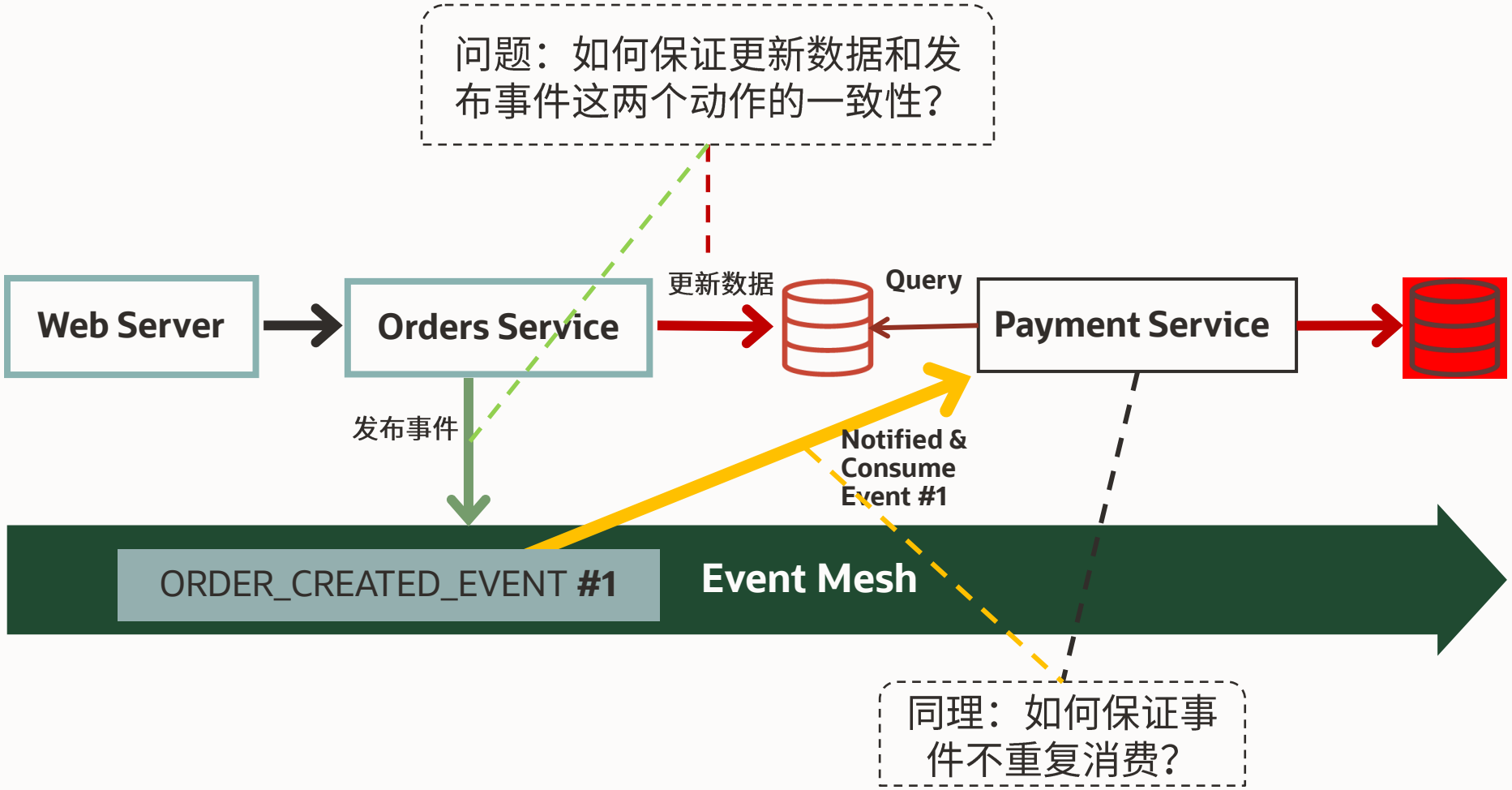
思考

- 发布事件通常会用到事件中间件，典型的如Kafka
- Transactional Outbox。参与者更新完本地数据后，如果发布事件失败怎么办（即更新本地数据和发布事件不在同一个事务中)?

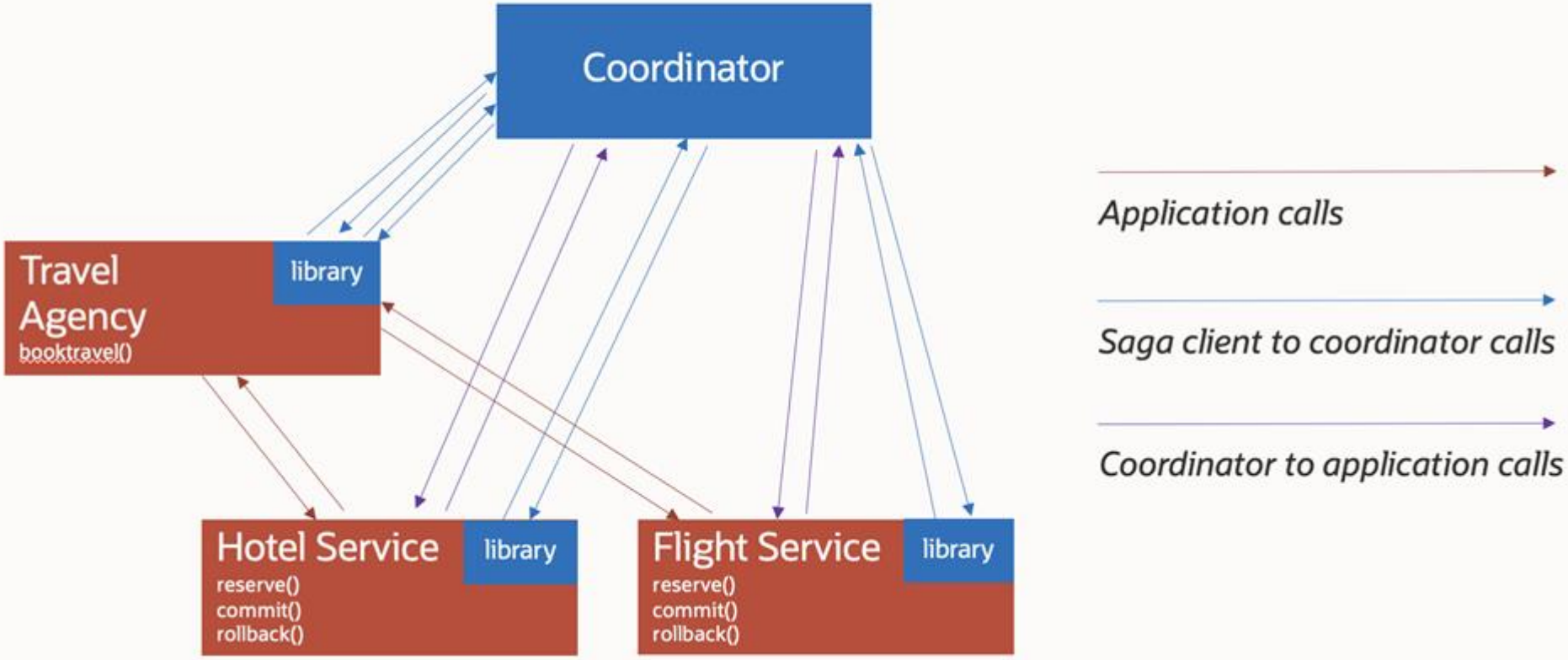


Source: Denis Rosa, Couchbase

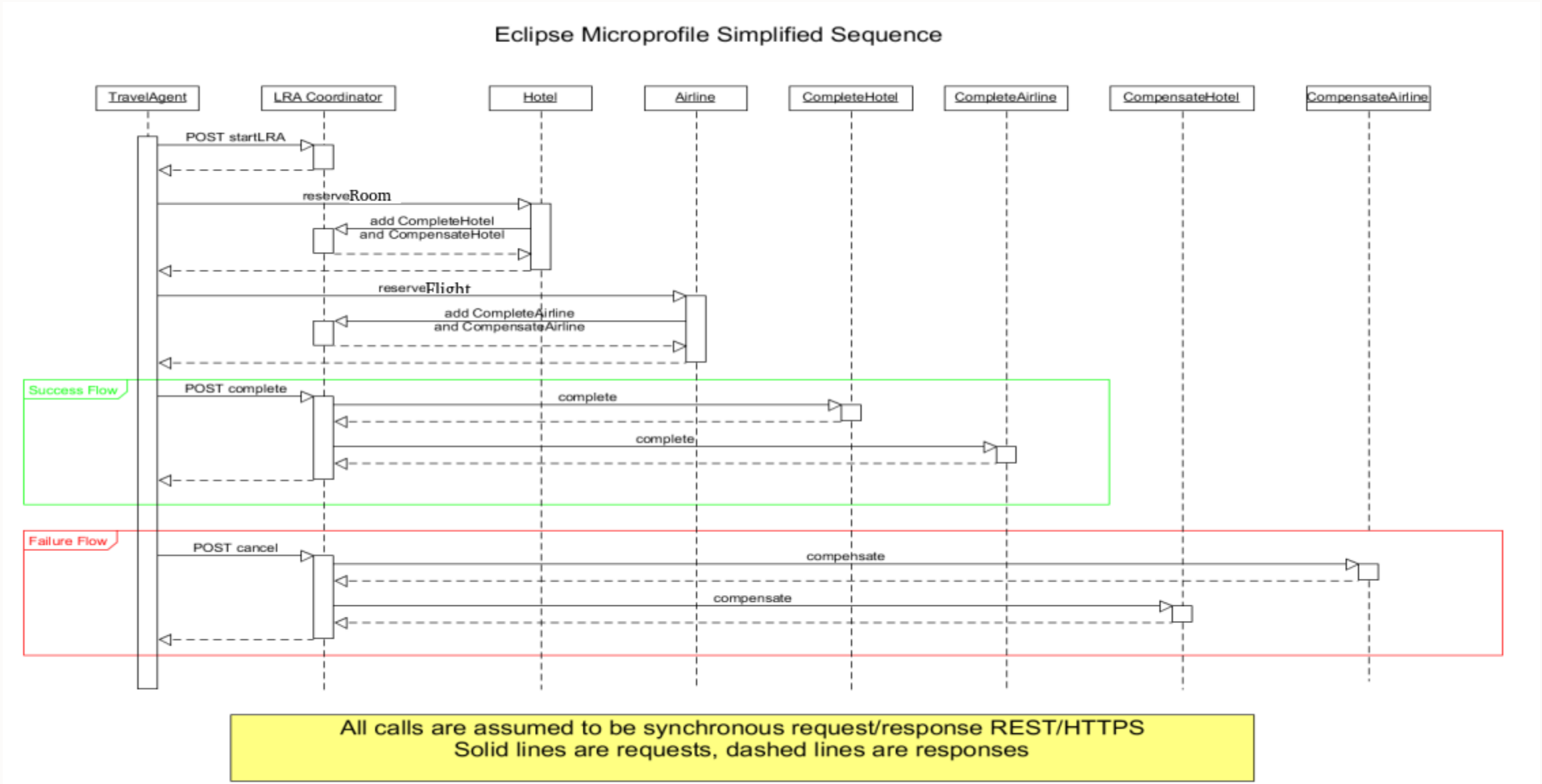
Choreography (无中央编排器)模式的思考



SAGA事务中央编排器模式(Orchestrator / Coordinator)



MicroProfile LRA(Long Running Action)事务处理流程



LRA参与者样例

```
@PUT
@LRA(LRA.Type.REQUIRES_NEW) ❶
@Path("start-example")
public Response startExample(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId, ❷
                             String data) {
    if (data.contains("BOOM")) {
        throw new RuntimeException("BOOM 💥"); ❸
    }

    LOGGER.info("Data " + data + " processed 🏭");
    return Response.ok().build(); ❹
}

@PUT
@Complete ❺
@Path("complete-example")
public Response completeExample(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {
    LOGGER.log(Level.INFO, "LRA id: {0} completed 🎉", lraId);
    return LRAResponse.completed();
}

@PUT
@Compensate ❻
@Path("compensate-example")
public Response compensateExample(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {
    LOGGER.log(Level.SEVERE, "LRA id: {0} compensated 🚑", lraId);
    return LRAResponse.compensated();
}
```

1. REQUIRES_NEW: 开启LRA事务
2. 由事务协调器(Coordinator)分配的全局事务ID
3. 处理出错，向事务协调器发送补偿动作信令
4. 处理成功，由事务协调器发送完成动作信令
5. @Complete: 由事务协调器调用的完成动作逻辑实现
6. @Compensate: 由事务协调器调用的补偿动作逻辑实现

@LRA - 选项/属性

选项/属性	描述
@LRA	LRA事务上下文
REQUIRED	如果存在LRA事务上下文，则加入现有的LRA事务，否则，新开一个LRA事务
REQUIRES_NEW	新开一个LRA事务
MANDATORY	参与现有的LRA事务，如果LRA上下文不存在，则报错
SUPPORTS	参与现有的LRA事务，但如果LRA上下文不存在，则忽略
NOT_SUPPORTED	始终不参与LRA事务(不管LRA上下文存不存在)
NEVER	不能在LRA事务上下文中执行，否则直接报HTTP 412错误
NESTED	创建嵌套LRA子事务
timeLimit/timeUnit	事务最长时间限制，过期自动cancel (compensate)
End	方法成功执行后是否需要关闭(complete)LRA事务
cancelOn/cancelOnFamily	在什么HTTP状态码时触发cancel操作 (compensate)

LRA参与者样例

```
@PUT
@LRA(LRA.Type.REQUIRES_NEW) ❶
@Path("start-example")
public Response startExample(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId, ❷
                             String data) {
    if (data.contains("BOOM")) {
        throw new RuntimeException("BOOM 💥"); ❸
    }

    LOGGER.info("Data " + data + " processed 🏭");
    return Response.ok().build(); ❹
}

@PUT
@Complete ❺
@Path("complete-example")
public Response completeExample(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {
    LOGGER.log(Level.INFO, "LRA id: {0} completed 🎉", lraId);
    return LRAResponse.completed();
}

@PUT
@Compensate ❻
@Path("compensate-example")
public Response compensateExample(@HeaderParam(LRA_HTTP_CONTEXT_HEADER) URI lraId) {
    LOGGER.log(Level.SEVERE, "LRA id: {0} compensated 🚑", lraId);
    return LRAResponse.compensated();
}
```

1. REQUIRES_NEW: 开启LRA事务
2. 由事务协调器(Coordinator)分配的全局事务ID
3. 处理出错，向事务协调器发送补偿动作信令
4. 处理成功，由事务协调器发送完成动作信令
5. @Complete: 由事务协调器调用的完成动作逻辑实现
6. @Compensate: 由事务协调器调用的补偿动作逻辑实现

LRA - 接口

注释	描述
@Compensate	事务取消时需要执行的补偿方法。通常由事务协调器自动触发。一般用HTTP PUT表示幂等操作。
@Complete	事务正常完成时执行的方法。通常由事务协调器自动触发。一般用HTTP PUT表示幂等操作。
@Status	事务协调器通过此方法获取参与者当前的状态，用以决定在@Complete / @Compensate 动作失败时 是否需要重试。当@Complete/@Compensate方法本身不幂等时此方法很重要。
@Forget	complete/compensate出错(500)或异步(202)处理情况／内嵌LRA，status超时仍未补偿完成／上层LRA关闭时告知已完成的子LRA。一般用HTTP DELETE表示
@Leave	告诉事务协调器此参与者离开了当前的LRA事务，因此当LRA全局事务结束时，此参与者的@Complete/@Compensate方法不会执行。通常由外部触发。
@AfterLRA	当前LRA事务完成后，由事务协调器自动触发。一般用HTTP PUT表示。
(HEADER参数) @HeaderParam	LRA_HTTP_CONTEXT_HEADER：事务ID (lraId) LRA_HTTP_PARENT_CONTEXT_HEADER：上层事务ID (有嵌套情况下) LRA_HTTP_ENDED_CONTEXT_HEADER：已完成的事务ID (@AfterLRA用到它)

LRA状态模型

事务发起者角度(全局):

Active：LRA已经创建，尚没执行关闭/取消动作

Cancelling: 正触发取消LRA的动作

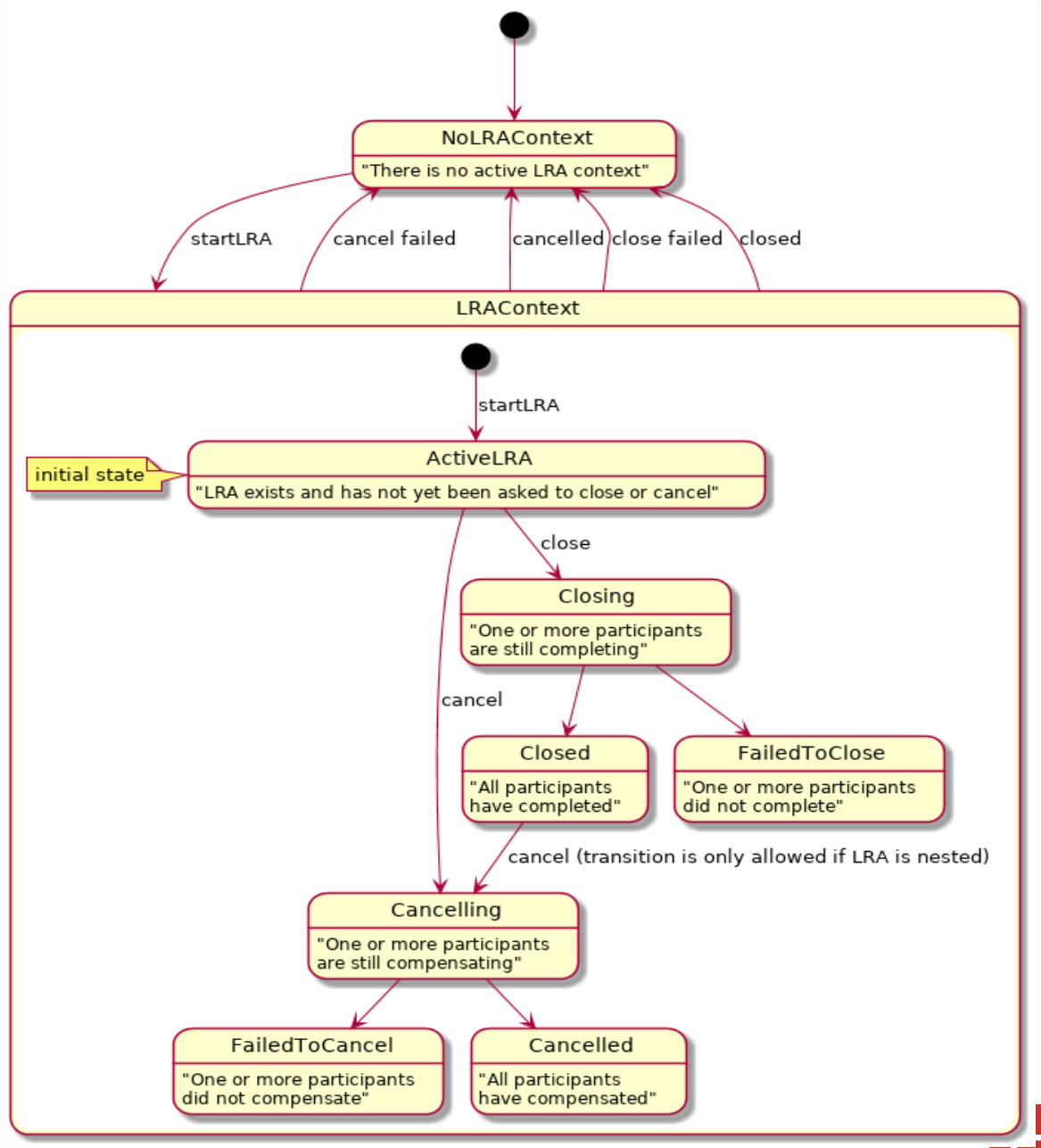
Cancelled：成功取消了当前LRA事务(所有参与者都成功完成了补偿逻辑)

FailedToCancel：取消LRA出现了问题(至少一个参与者执行补偿逻辑失败了)

Closing：正触发关闭LRA的动作

Closed：成功关闭了当前LRA事务(所有参与者都成功执行了完成逻辑)

FailedToClose：关闭LRA出现了问题(至少一个参与者执行完成逻辑失败了)



Source: Eclipse MicroProfile LRA

LRA状态模型

事务参与者角度(具体服务)

Active：已加入LRA，尚没执行完成/补偿动作

Compensating: 正触发补偿动作

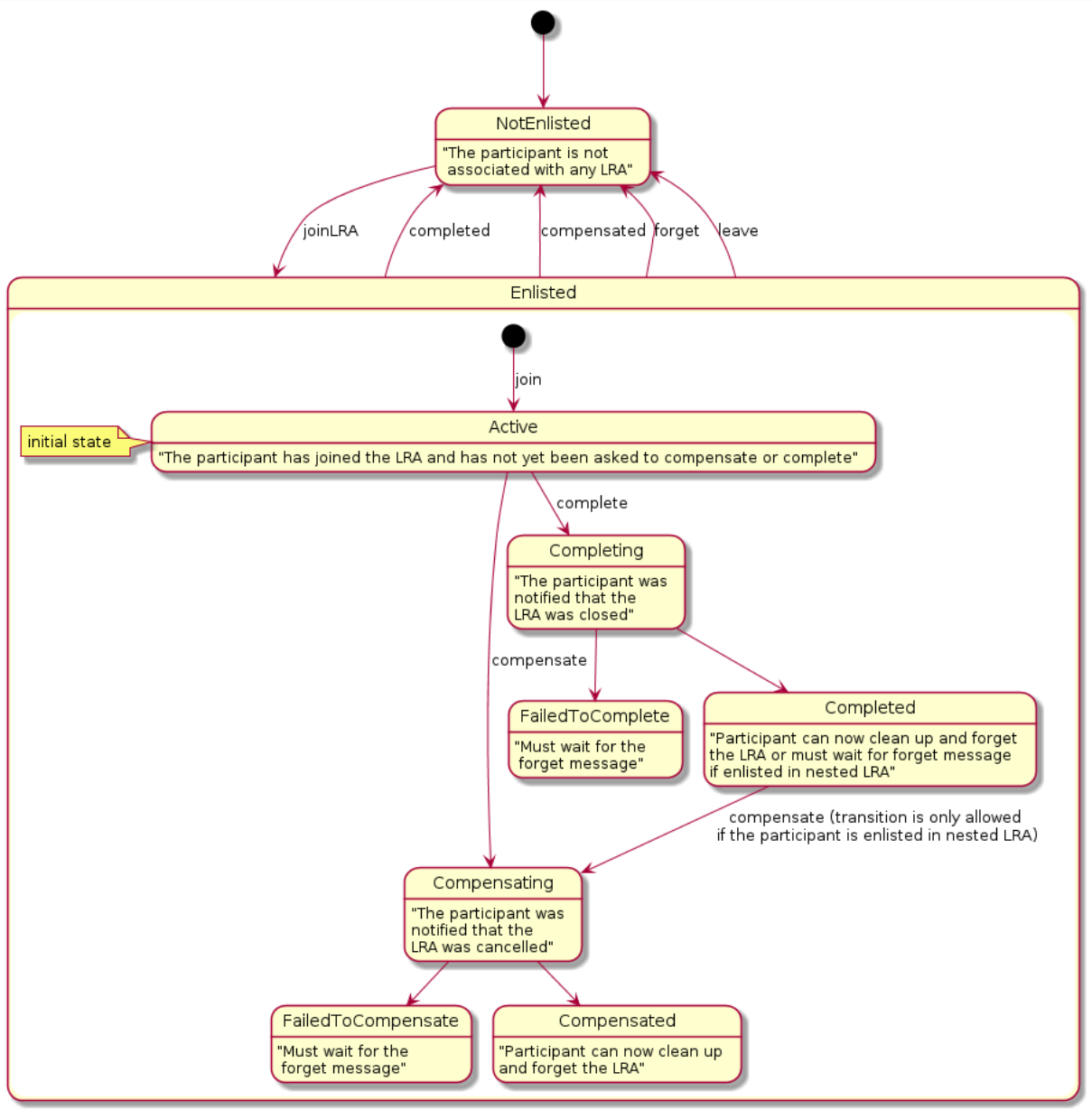
Compensated：成功执行了补偿动作

FailedToCompensate：补偿失败了

Completing：正触发完成动作(清理)

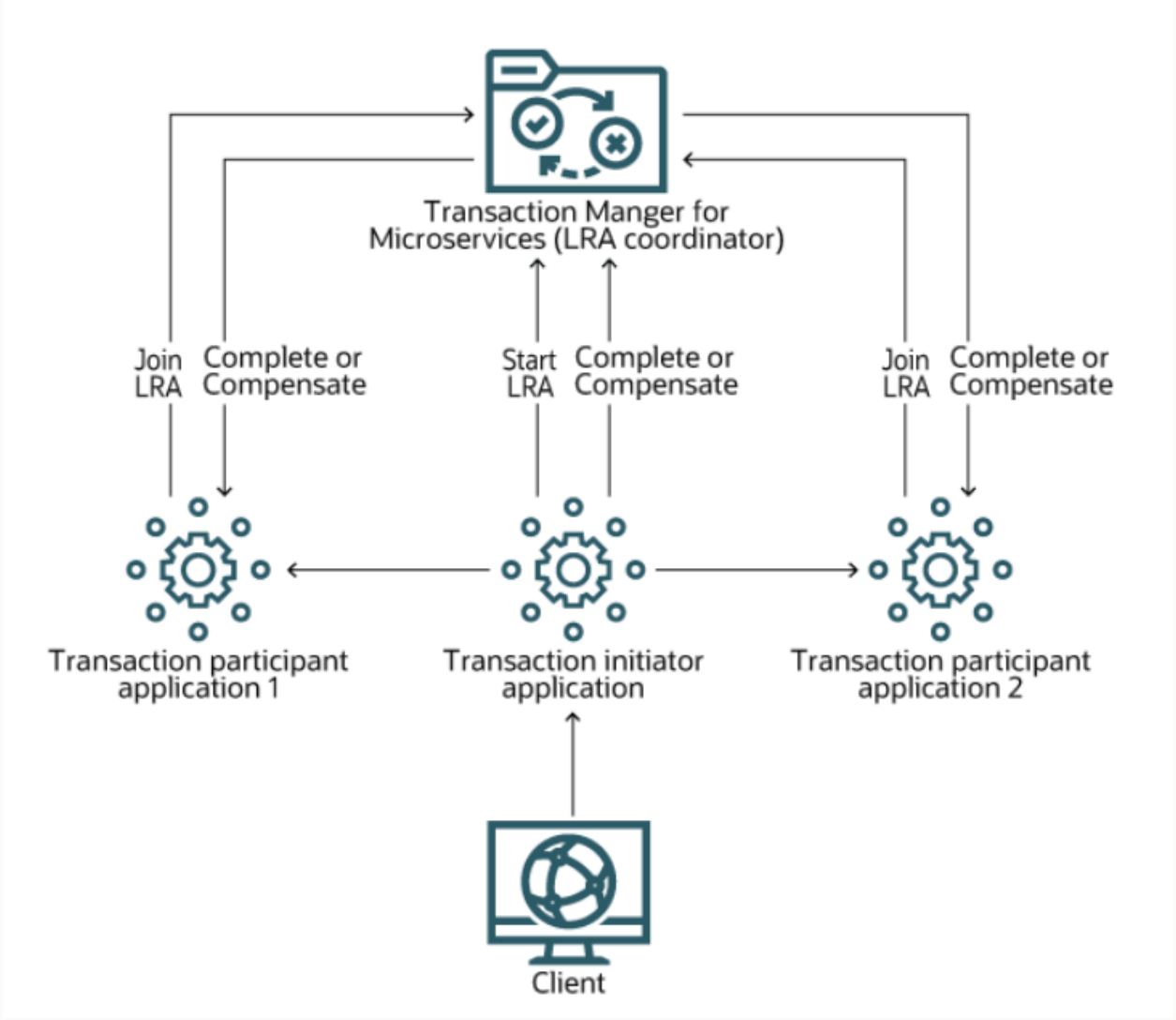
Completed：成功执行了完成逻辑

FailedToComplete：执行完成逻辑失败了



Source: Eclipse MicroProfile LRA

SAGA / LRA 小结



定义

- 各参与者都执行本地事务。一个LRA就是一连串的这样的本地事务组成的一个全局事务。如果中途有任意参与者失败，就执行**补偿**动作。

优点

- 本地事务，无全局锁，性能好
- 支持长时间跨度的事务
- 1PC
- ACID中，不支持 ”I”

不足

- 最终一致性，存在不一致的中间状态可能
- 有开发工作量，需要编写代码实现 具体业务(事务)的完成/补偿操作

场景

- 长流程事务(可能包括人工交互)，能接受中间不一致性状态。

分布式事务总结

特性	XA	TCC	Saga (LRA)
一致性	最强一致性	强一致性	最终一致性
原子性	✓	✓	✓
隔离性	数据层面隔离	业务上隔离	不提供
持久性	✓	✓	✓
锁	全局(数据源)锁	本地事务锁+预定资源(业务锁)	本地事务锁
提交方式	两阶段	两阶段	一阶段
实现主体	数据源(驱动)支持	业务层实现两阶段逻辑	业务层实现补偿逻辑
适应范围	尽量简短，范围尽量小	短流程	长流程
企业集成	不适用	难适用 (要求高、成本高)	比较适用
使用便捷性	更方便	一般	一般
性能	一般	较好	更好
人工介入(运维)	基本不用 (除非发生 HeuristicMixedException)	最终不能成功时需要	最终不能成功时需要

议程

背景

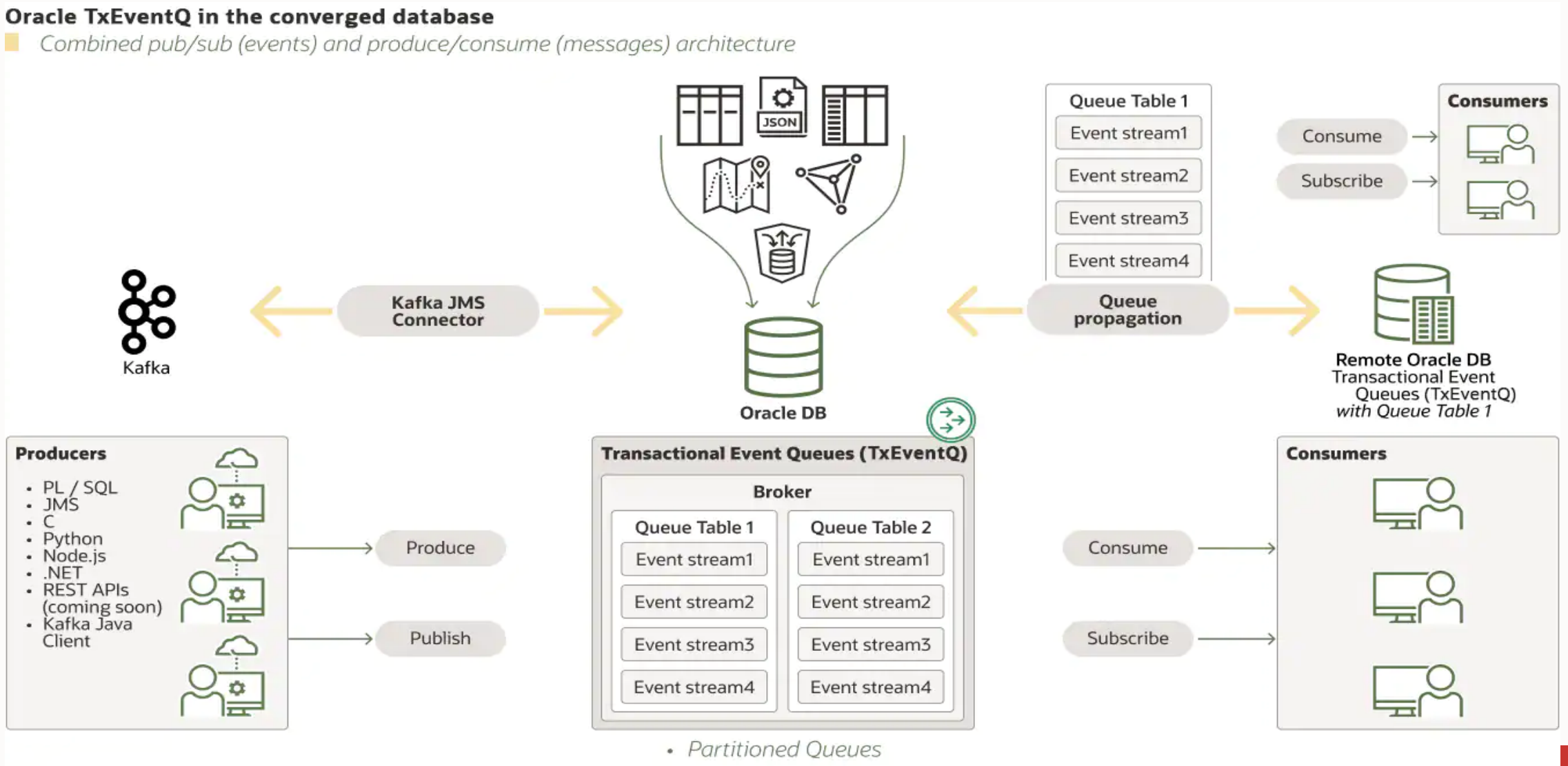
分布式事务协议

- XA / LLR
- TCC
- Saga (LRA)

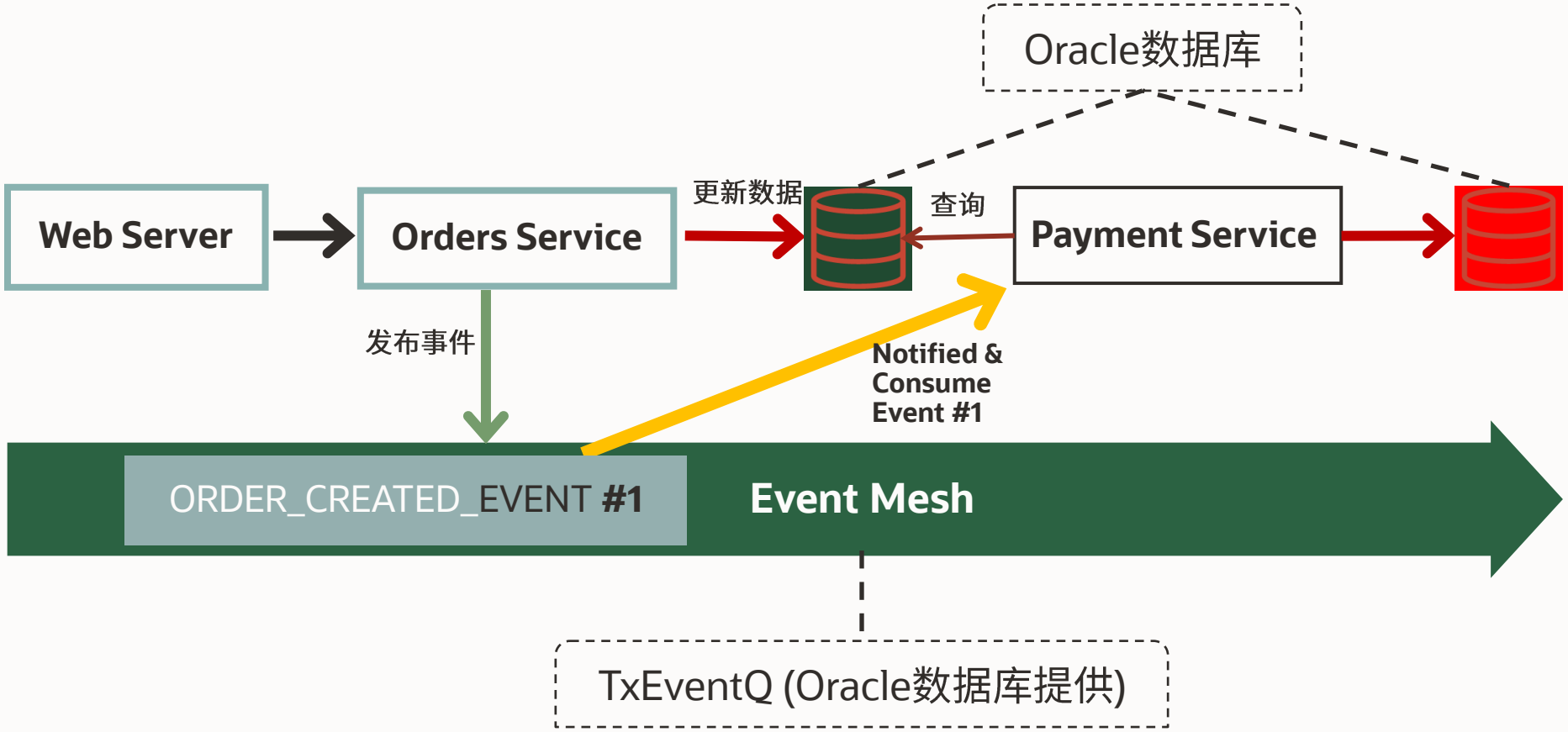
Oracle数据库Saga支持

Oracle 微服务事务中间件

Oracle数据库对Saga Choreography(无中央编排器)模式的支持 - Oracle TxEventQ



Choreography & Oracle TxEventQ - 基于事务的事件发布与消费



- 更新数据表数据与发布事件这两个动作处于同一个本地数据库事务中，保证了原子性(一致性)
- 同时事务保证了真正意义上的exactly once特性 (事件既不会重复发布，也不会重复消费)

```
    try {
        TopicConnectionFactory q_cf = AQjmsFactory.getTopicConnectionFactory(dataSource);
        TopicConnection q_conn = q_cf.createTopicConnection();
        session = q_conn.createTopicSession(true, Session.CLIENT_ACKNOWLEDGE);
        OracleConnection jdbcConnection = ((OracleConnection)((AQjmsSession) session).getDBConnection());
        System.out.println("OrderServiceEventProducer.updateDataAndSendEvent activespan ecid=" + activeSpan);
        short seqnum = 20;
        String[] metric = new String[OracleConnection.END_TO_END_STATE_INDEX_MAX];
        metric[OracleConnection.END_TO_END_ACTION_INDEX] = "orderservice_action_placeOrder";
        metric[OracleConnection.END_TO_END_MODULE_INDEX] = "orderservice_module";
        metric[OracleConnection.END_TO_END_CLIENTID_INDEX] = "orderservice_clientid";
        metric[OracleConnection.END_TO_END_ECID_INDEX] = spanIdForECID; //for log to trace
        activeSpan.setBaggageItem("ecid", spanIdForECID); //for trace to log
        jdbcConnection.setEndToEndMetrics(metric,seqnum);//  todo instead use  conn.setClientInfo();
        System.out.println("updateDataAndSendEvent jdbcConnection:" + jdbcConnection + " activeSpan:" + activeSpan + "
        Order insertedOrder = insertOrderViaSODA(orderid, itemid, deliverylocation, jdbcConnection);
        if (OrderResource.crashAfterInsert) System.exit(-1);
        System.out.println("updateDataAndSendEvent insertOrderViaSODA complete about to send order message...");
        Topic topic = ((AQjmsSession) session).getTopic(OrderResource.orderQueueOwner, OrderResource.orderQueueName);
        System.out.println("updateDataAndSendEvent topic:" + topic);
        TextMessage objmsg = session.createTextMessage();
        TracingMessageProducer producer = new TracingMessageProducer(session.createPublisher(topic), orderResource.getTracer());
        objmsg.setIntProperty("Id", 1);
        objmsg.setIntProperty("Priority", 2);
        String jsonString = JsonUtils.writeValueAsString(insertedOrder);
        objmsg.setText(jsonString);
        objmsg.setJMSPriority(2);
        producer.send(topic, objmsg, DeliveryMode.PERSISTENT, 2, AQjmsConstants.EXPIRATION_NEVER);
//          publisher.publish(topic, objmsg, DeliveryMode.PERSISTENT,2, AQjmsConstants.EXPIRATION_NEVER);
        session.commit();
        System.out.println("updateDataAndSendEvent committed JSON order in database and sent message in the same tx with payload:" + jsonString);
        return topic.toString();
    } catch (Exception e) {
        System.out.println("updateDataAndSendEvent failed with exception:" + e);
        if (session != null) {
            try {
                session.rollback();
            } catch (JMSException e1) {
                System.out.println("updateDataAndSendEvent session.rollback() failed:" + e1);
```

获取数据库连接

更新订单数据库

发布事件

更新数据库和发布事件在同一事务中

Oracle数据库对Saga Orchestrator/Coordinator(中央编排器)模式的支持 - Osaga

Without Osaga support

```
// startup logic : requires devoted timeout and recovery processes
//                 and maintaining persistent log for saga records…
   readRecoveryLogs(connection);
   // recovery processing
   startRecoveryProcessing (connection);
   startExpirationProcessing (connection);

// individual saga logic…
   String sagaId = beginSaga(connection);
   persistSagaRecord(connection, sagaId);
   persistFlightParticipantRecord(connection, sagaId);
   reserveFlight(connection, sagaId);
   persistHotelParticipantRecord(connection, sagaId);
   reserveHotel(connection, sagaId);

// the AQ transactional outbox pattern is used here by
//  sending the saga events and state change in the same local transaction
   if (isCommit) {
      connection.setAutoCommit(false);
      sendCommitEventToAllParticipants(connection, sagaId);
      setStateToCommittedOnParticipants(connection, sagaId);
      purgeSagaAndParticipantEntries(connection, sagaId);
      connection.commit();
   } else {
      connection.setAutoCommit(false);
      sendRollbackEventToAllParticipants(connection, sagaId);
      setStateToRolledbackOnParticipants(connection, sagaId);
      purgeSagaAndParticipantEntries(connection, sagaId);
      connection.commit();
   }
```

Advantages With Osaga Support

```
// no startup logic required.

// individual saga logic…
beginOSaga (); //Osaga.beginSaga()
reserveFlight(); //provide AQjmsSagaMessageListener implementation
reserveHotel(); //provide AQjmsSagaMessageListener implementation

if (isCommit) commitSaga(); //Osaga.commitSaga();
else  rollbackSaga(); //Osaga.rollbackSaga();
```

Osaga – 基于MicroProfile LRA规范，在Oracle库内实现，除基本功能外，还拥有数据库本身的一些特性。

OSaga 特性 (Oracle 23C)

Without Osaga or LRA support	With LRA	With Oracle DB Saga	Without LRA or Oracle DB Saga
Automatic propagation of saga id and participant enlistment	X	X	
Automatic coordination of completion protocol (commit/rollback) from coordinator to all participants.	X	X	
Automatic timeout/rollback logic	X	X	
Automatic recovery logic	X	X	
REST support	X	X	
Messaging support	*future	X	
Automatic recovery state maintained in participants		X	
Automatic HA		X	
Automatic Compensating Data		*future	

议程

背景

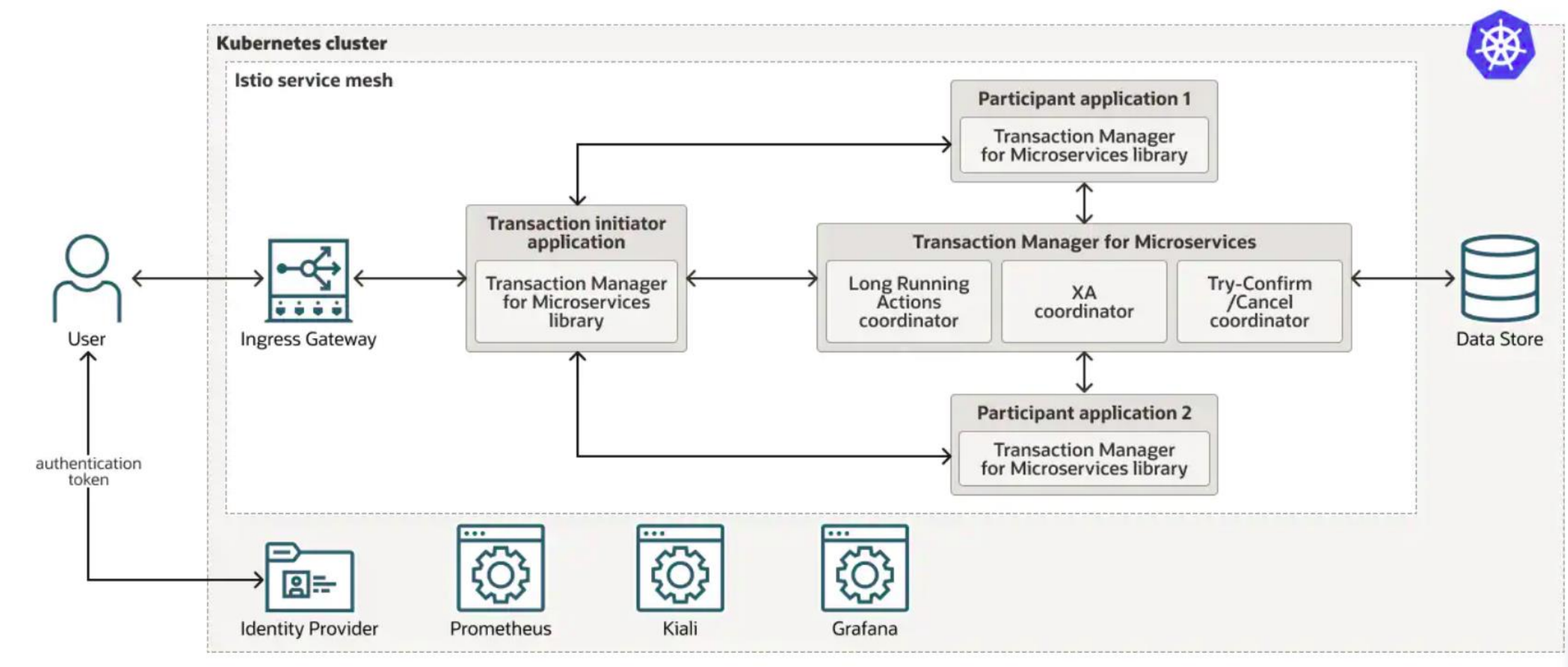
分布式事务协议

- XA / LLR
- TCC
- Saga (LRA)

Oracle数据库Saga支持

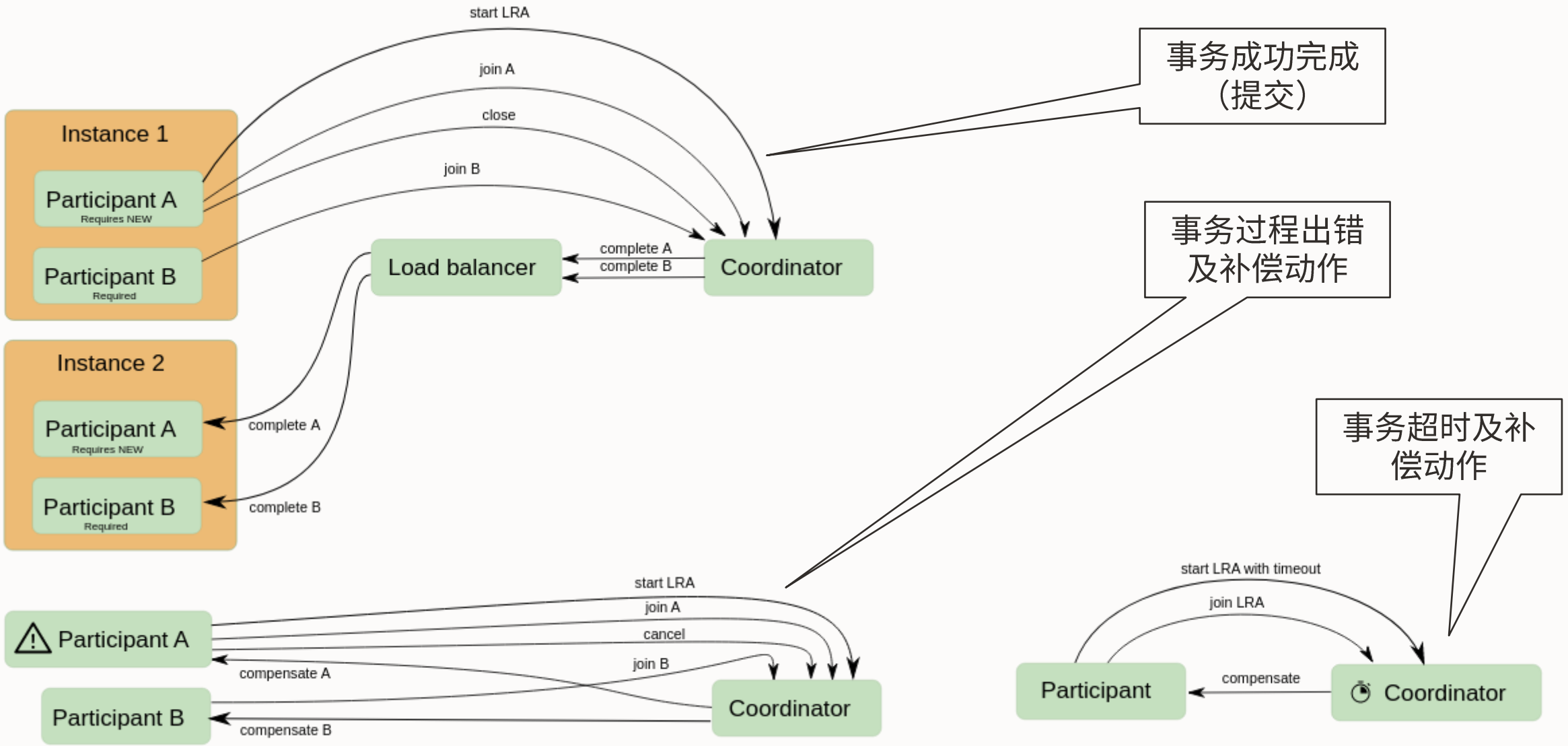
Oracle 微服务事务中间件

Oracle事务中间件MicroTx (Orchestrator / Coordinator)



Oracle推出的分布式事务中间件，支持XA (以及LLR), TCC 和 LRA协议，基于Java，客户端支持Spring, Helidon, NodeJs, PLSQL等。完全免费。官网：https://www.oracle.com/database/transaction-manager-for-microservices/

MicroTx 事务协调器 (Orchestrator)



MicroTx 小结

本身微服务部署、云原生

功能丰富

- 支持XA，TCC，LRA以及XA扩展协议LLR。LRA基于SAGA Orchestrator/Coordinator模式，遵循MicroProfile LRA规范实现。

提高开发人员生产力

- 消除了开发人员需要编写调度逻辑(及测试)来补偿失败的交易，只需要关注失败交易的业务/数据补偿，从而提高了生产力。

利用现有资产、开发人员框架

- 开发人员可以继续使用现有的工具和框架；MicroTx只需要对现有应用程序进行一些更改。

基于行业标准

- MicroTx使用行业标准事务协议（如XA、LRA和TCC）实现数据一致性，从而降低供应商锁定的风险。

多语言系统中实现一致性

- 使用MicroTx，用Java、TypeScript、C/C++等编写的多语言微服务和框架可以参与全局分布式事务。

数据库和云讲座群

20-20

甲骨文云技术公众号

技术专家1V1深入交流