redhat

DISTRIBUTED TRANSACTIONS

SYSTEM INTEGRATION WITH JBOSS

Vaclav Chalupa & Viliam Kasala | QE at RedHat

WHAT IS A TRANSACTION?

In computer programming, a transaction means a sequence of operations (database updates) where data integrity is ensured.

ACID

ATOMICITY All, or Nothing

CONSISTENCY From one valid state to another valid state

ISOLATION Transactions do not affect each other

Isolation levels (relaxation):
READ_UNCOMMITED < READ_COMMITED < REPEATEBLE_READ < SERIALIZABLE

DURABILITY Stored permanently

DISTRIBUTED TRANSACTION

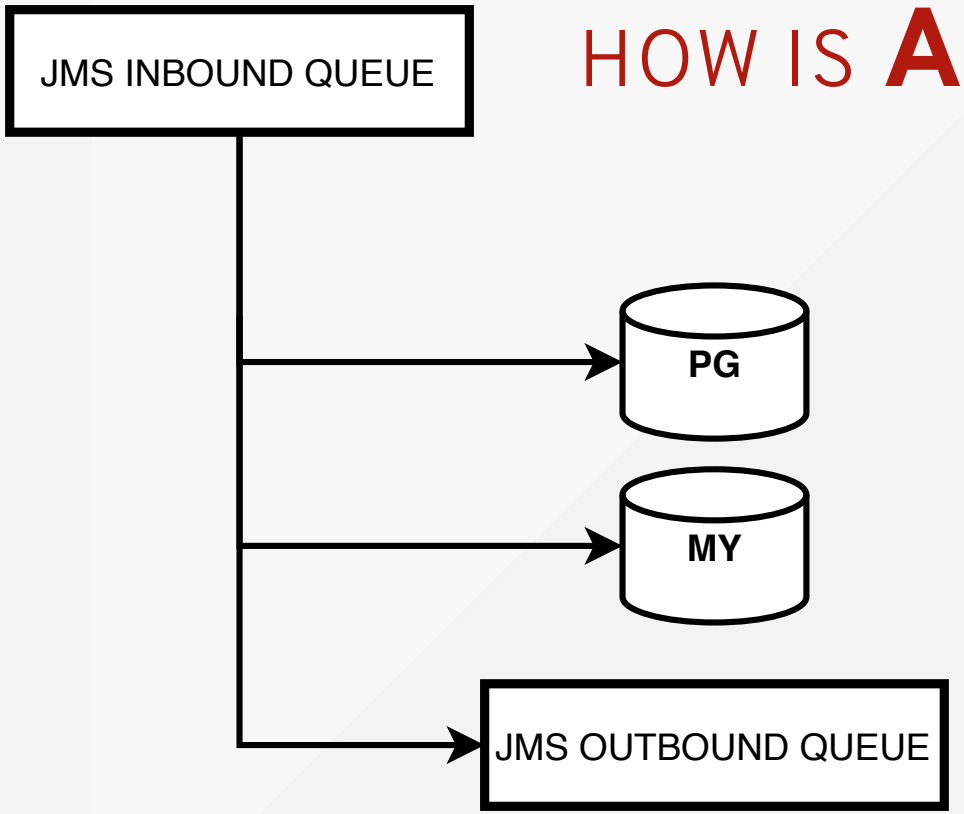
A transaction in which **two and more transactional resources** are involved

DT, as any other transactions, **must have all four ACID properties**

Transactional resources?

- Databases
- JMS
- EIS (Enterprise Integration Systems supporting transactions)

DT EXAMPLE



HOW IS ACID ACHIEVED?

TRANSACTION MANAGER

TRANSACTION COORDINATOR

Transaction manager is component that is **responsible for coordinating of distributed transactions.**

Transaction manager uses **two-phase commit protocol** to perform **atomic distributed transactions**.

2PC

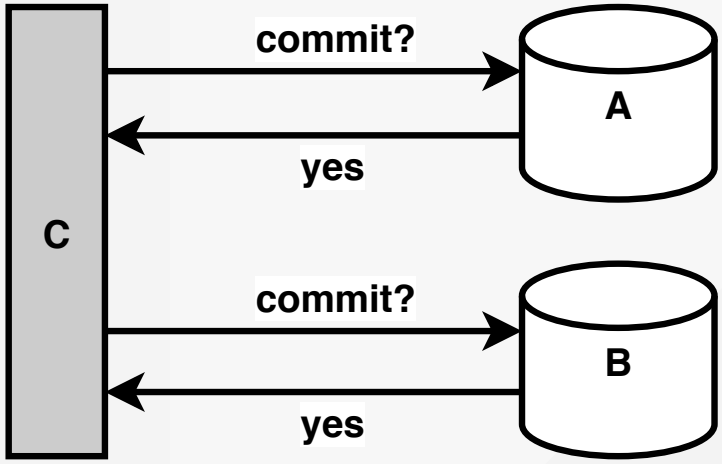
TWO-PHASE COMMIT PROTOCOL

An algorithm ensuring correct completion of a distributed transaction = Atomicity

It coordinates all participated transactional resources within distributed transaction on whether to **commit** or **rollback** (abort).

2PC | PHASE I

VOTING/PREPARE PHASE



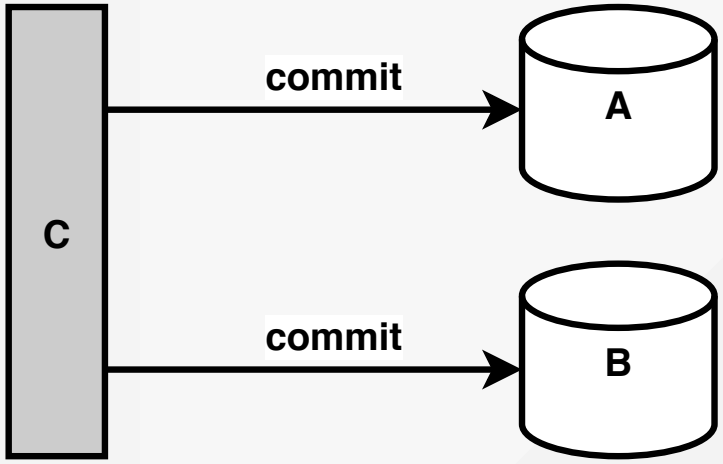
Answer **NO** or **no answer** causes **rollback** of DT.

If the transaction **will commit**, the transaction coordinator **records the decision** on stable storage, and the protocol **enters phase II**.

If the transaction **will abort** all participant are **informed** about this decision **too**.

2PC | PHASE II

COMMIT PHASE

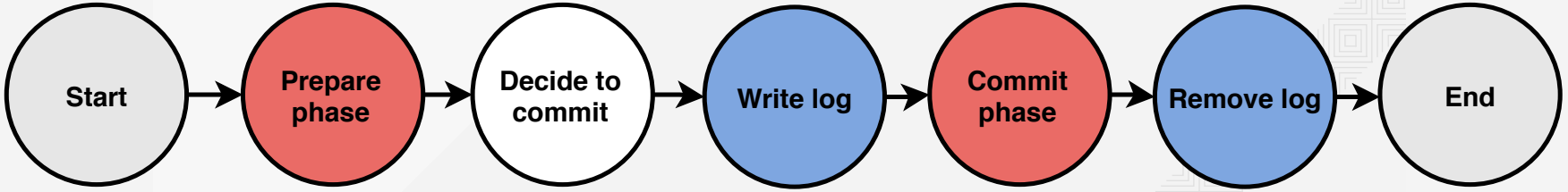


TRANSACTION LOG

Transaction log is used for transaction recovery in case of failure.

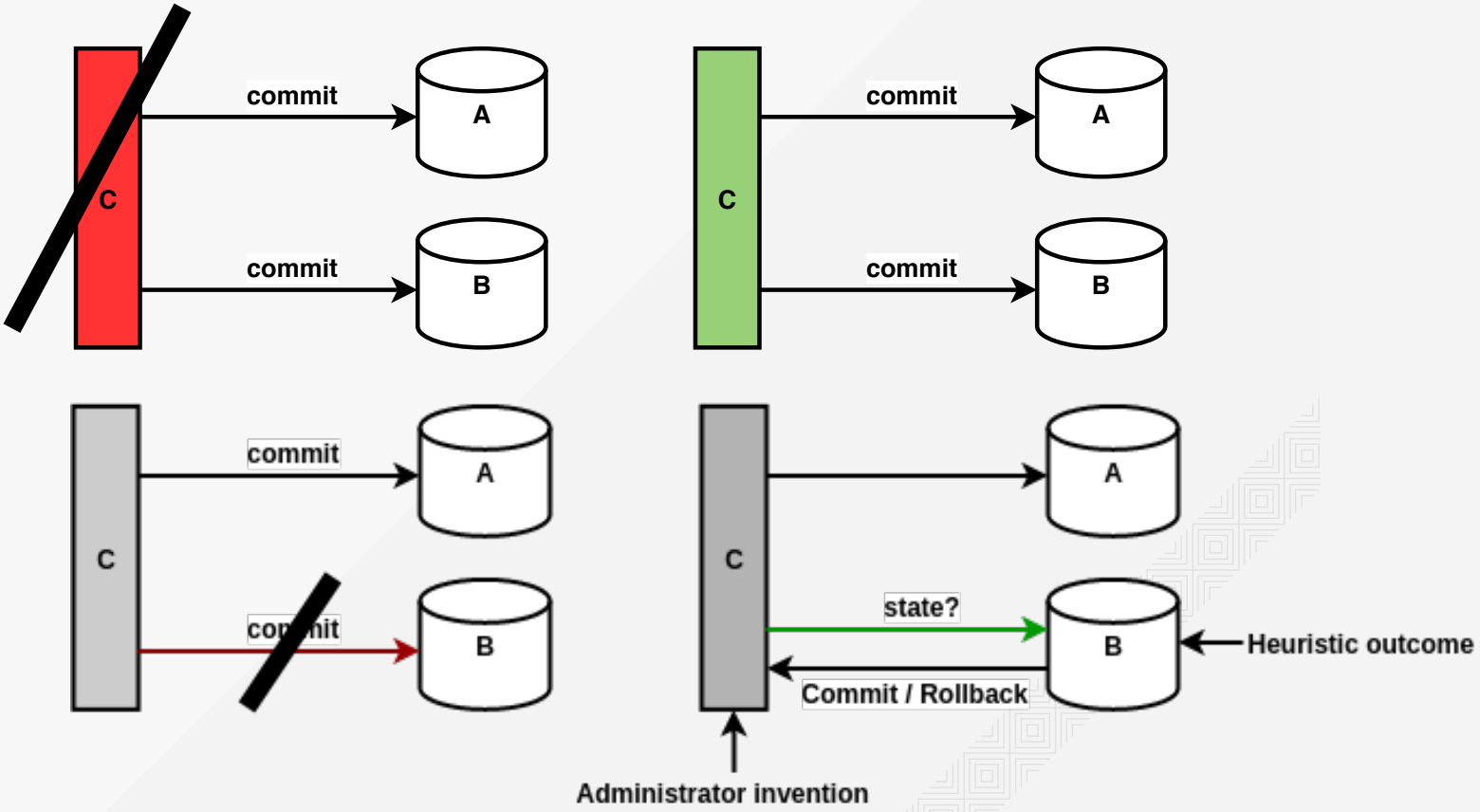
If a distributed transaction will commit, **transaction coordinator writes the decision to log.**

Each **participant** must have own **transaction log** to write commit decision.



FAILURE RECOVERY

COORDINATOR FAILURE | NETWORK / PARTICIPANT FAILURE



2PC OPTIMIZATIONS

ONE-PHASE

If there is only one involved transactional resource, transaction manager can use simply commit.

LAST RESOUCE COMMIT (GAMBIT)

It is possible to enlist one resource which is not two-phase commit aware.

X/OPEN XA

"EXTENTED ARCHITECTURE"

The Open Group defines standard for Distributed Transaction Processing **(DTP) = standard for distributed transactions**

The standard describes how the transaction manager must behave to coordinate distributed transaction and what resource managers of transactional resources must do to support transactional access.

XA Transaction = **Distributed Transaction**

XA Resource = **Transactional Resource** (Participant)

JTA

JAVA TRANSACTION API

JTA is java implementation of X/OPEN XA specification

JTA specifies standard Java interfaces between a **transaction manager** and the parties involved in a distributed transaction system: the **resource manager**, the **application server** and the **transactional applications**.

- High-level application interface for transaction boundaries demarcation
- High-level transaction manager interface used by container to control transactions

JTA is a specification developed under the Java Community Process as JSR 907.

JTS

JAVA TRANSACTION SERVICE

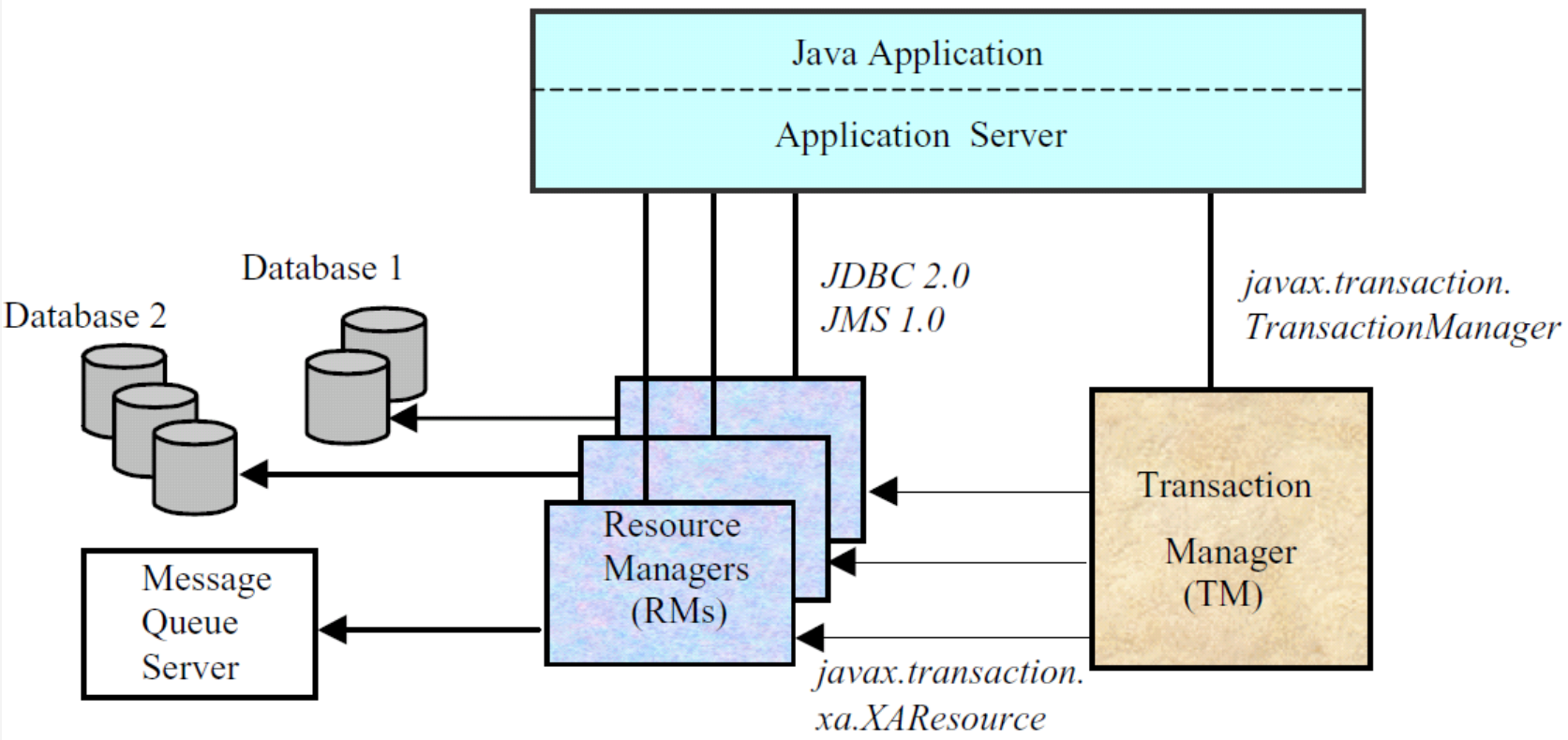
JTS is implementation of another specification: **Object Transaction Service (OTS)** specified by **OMG**

JTS specifies an implementation of a transaction manager that support JTA specification = JTS can be used by JTA

JTS uses the standard CORBA ORB/TS interfaces and Internet Inter-ORB Protocol (IIOP) for transaction context propagation between JTS transaction managers.

JTA is high-level API
JTS is low-level API

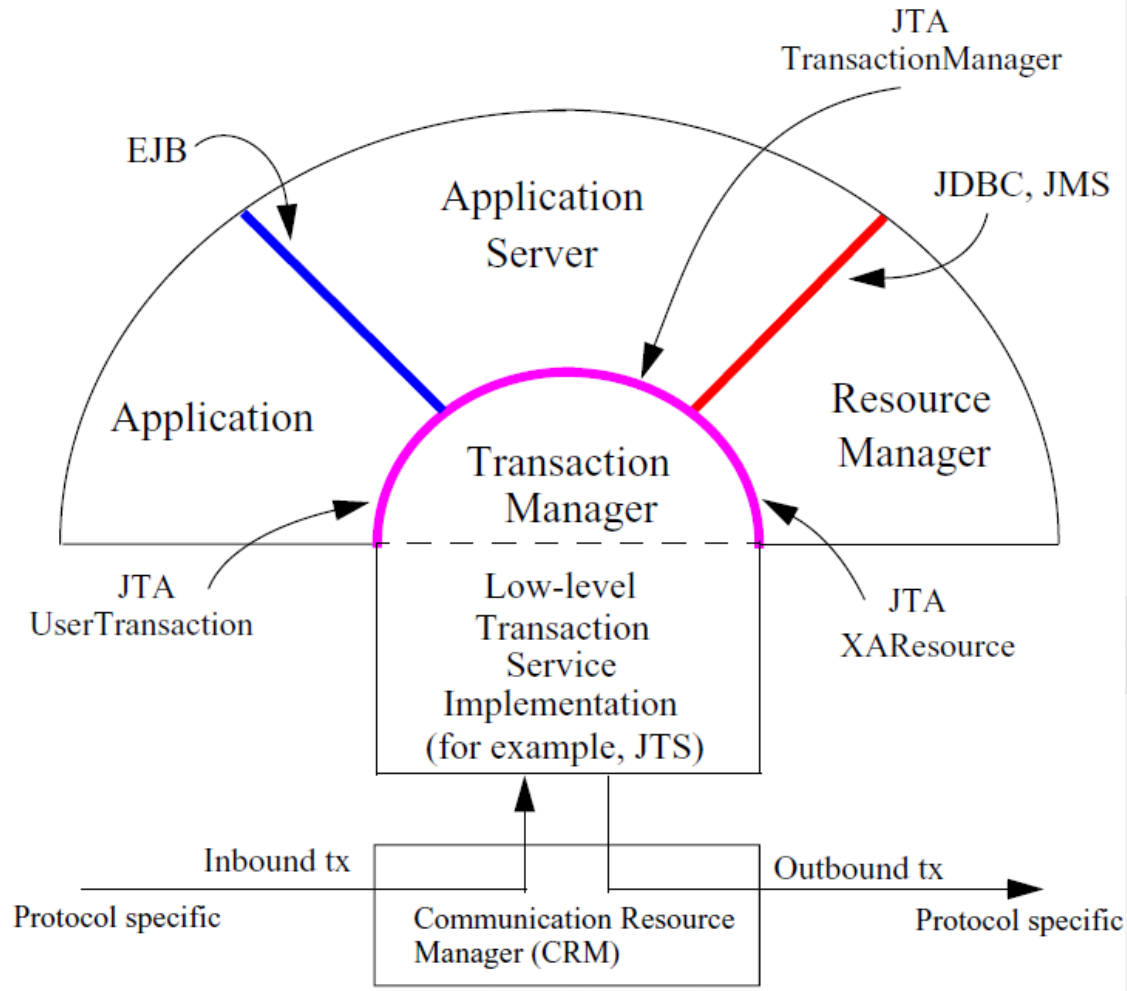
X/OPEN XA ARCHITECTURE



JTA INTERFACES

```
javax.transaction.UserTransaction
javax.transaction.TransactionManager
javax.transaction.Transaction
javax.transaction.Synchronization

javax.transaction.xa.XAResource
javax.transaction.xa.Xid
```

JTA ARCHITECTURE IN JAVA EE



USER TRANSACTION

JAVAX.TRANSACTION.USERTRANSACTION

Used by Java EE applications to programmatically demarcate transactions

Obtained via JNDI (java:jboss/UserTransaction) or using @Inject annotation (Java EE)

Operations:

begin() creates a new transaction and associate it with the current thread

commit() completes the transaction associated with current thread

rollback() rollback the transaction associated with current thread

setRollbackOnly() only possible outcome of the transaction of current thread is rollback

getStatus() obtain the status of the transaction associated with current thread

TRANSACTION MANAGER

JAVAX.TRANSACTION.TRANSACTIONMANAGER

`Used by Application container`

Obtained via JNDI - *java:jboss/TransactionManager*

`Operations:`

begin() creates a new transaction and associate it with the current thread

commit() completes the transaction associated with current thread

rollback() rollback the transaction associated with current thread

setRollbackOnly() only possible outcome of the transaction of current thread is rollback

int getStatus() obtain the status of the transaction associated with current thread

Transaction getTransaction() gets the transaction of the calling thread

Transaction suspend() suspend transaction of the calling thread

resume(Transaction t) resumes the transaction context of the calling thread with t

TRANSACTION

JAVAX.TRANSACTION.TRANSACTION

Represent transaction started by transaction manager

Control transaction outcome: commit, rollback, setRollbackOnly

Enlist/delist transactional resources to transaction

Register synchornization callbacks with transaction

`Operations:`

commit() completes the transaction associated with current thread

rollback() rollback the transaction associated with current thread

setRollbackOnly() only possible outcome of the transaction of current thread is rollback

enlistResource(XAResouce xar) enlist the specified resource to transaction

delistResource(XAResouce xar, int flag) delist the specified resource from transaction

registerSynchronization(Synchronization sync) registers synchronization

SYNCHRONIZATION

JAVAX.TRANSACTION.SYNCHRONIZATION

Callback interface to inform about transaction completion

Typically used by cache to update/invalidate its contents

Best-effort only, not guaranteed to be called in case of crash

Operations:

beforeCompletion()

afterCompletion()

XA RESOURCE

JAVAX.TRANSACTION.XA.XARESOURCE

Represents any object that supports one or two phase protocol to participate in transaction and can ensure ACID properties

- Database connection
- JMS connection

`Operations (called exclusively by TransactionManager):`

start(Xid xid, int flag) starts the work on behalf of transaction branch specified in XID

end(Xid xid, int flag) ends the work performed on behalf of transaction branch

prepare(Xid xid) informs resource manager that it should prepare work for commital

commit(Xid xid, boolean onePhase) commit

rollback(Xid xid) rollback

Xid[] recover(int flag) obtains a list of prepared transaction branches from RM

forget(Xid xid) tells the resource manager to forget about a heuristically completed transaction branch

XID

JAVAX.TRANSACTION.XA.XID

Unique identifier of transaction

Industry standard

- Fully portable across different Transaction Managers, allows creation of delegated transactions in case of hierarchical Transaction Managers

Consists of three components

- Format identifier – must be unique across all transaction systems
- Global transaction identifier
- Branch qualifier

ENTERPRISE JAVABEANS

Transaction demarcation beans in Java EE application

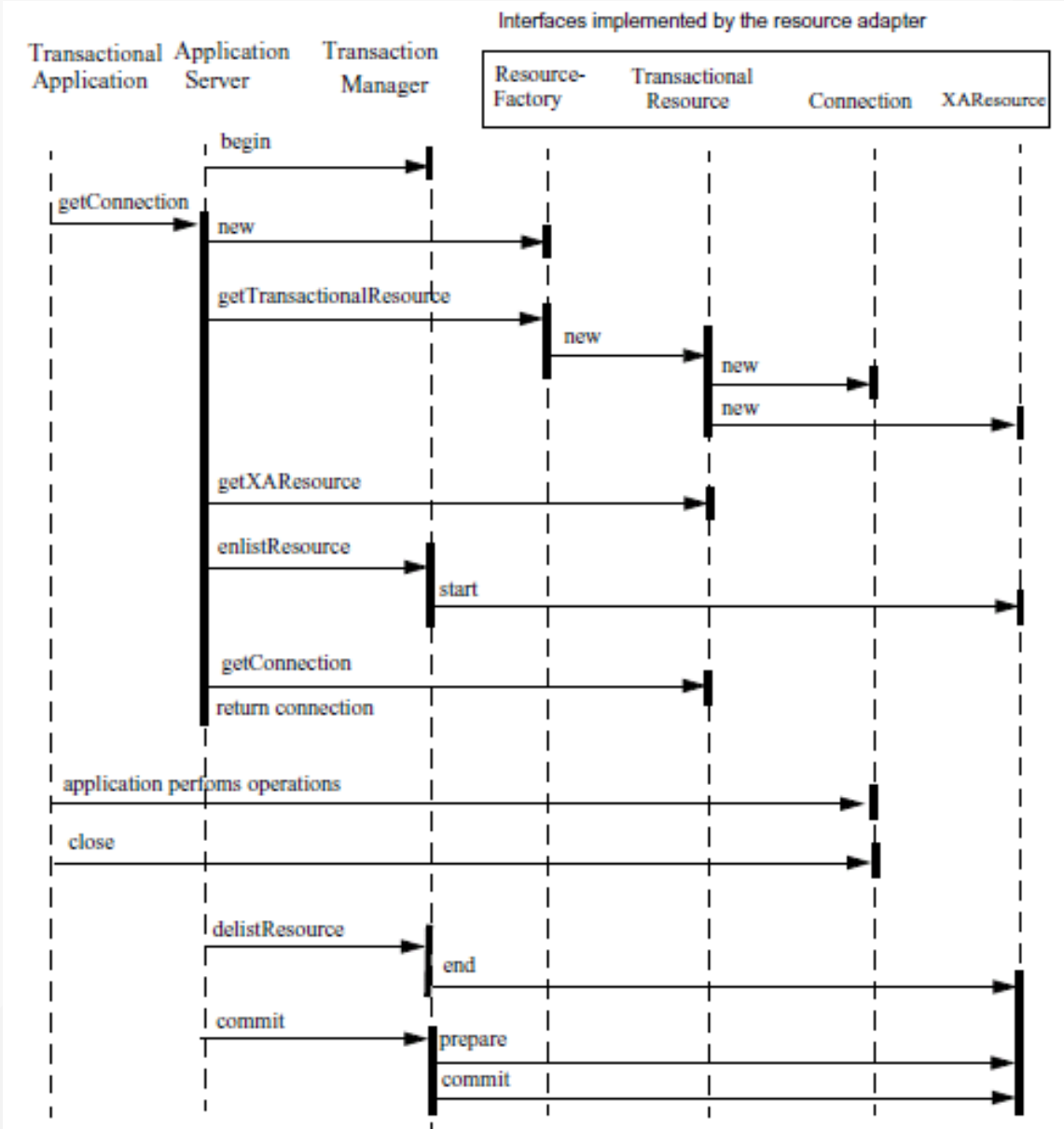
Demarcation models supported by container:

- **Container managed transaction**: @TransactionAttribute (REQUIRED, REQUIRES_NEW, ...)
- **Bean managed transactions**: @Inject UserTransaction

```
@Stateless
@TransactionManagement(CONTAINER)
public class UserDetailBean {

   @TransactionAttribute(REQUIRED)
   public void createUserDetail() {
      //create user details object
   }
}
```

```
@Stateless
@TransactionManagement(BEAN)
public class ExampleBean {

    @Inject private UserTransaction utx;

    public void foo() {
        // start a transaction
        utx.begin();
        // Do work
        // Commit it
        utx.commit();
}}
```

TRANSACTION FLOW



JBOSS FUSE GUIDES

XA TRANSACTION DECLARATION IN JBOSS FUSE

https://github.com/FuseByExample/esb-transactions

TRANSACTION GUIDE FOR JBOSS FUSE

https://access.redhat.com/documentation/en-us/red_hat_jboss_fuse/6.3/html-single/transaction_guide/

TRANSACTION AND WEB SERVICES

Common atomic transactions are short-running

= short atomic operations

For Web Services, we need **long-running** transaction which are **propagated** between communicating **Web Service**.

BEA, IBM, and Microsoft developed **WS-Coordinator** and WS-Tx specs.

WS-Tx was split to **WS-Atomic Transaction** ans **WS-BusinessActivity**.

WS-COORDINATOR

There is a need for a generic coordination infrastructure in a Web Service environment.

WS-Coordinator defines a framework that:

- allows different coordination protocols to be plugged-in
- coordinates work between clients, services and participants.

Consists of three main modules:

- **Activation Service** creates new coordinator & context
- **Registration Service** registers the participant with the coordinator
- **Context** contains information necessary to perform coordination, context is propagated between participants

WS-ATOMIC TRANSACTION

The specification:

- **extends** the coordinator context to create **transaction context**
- **augments** Activation and Registration services to **support atomic transaction** (emulates ACID)
- **adds/defines** two protocols - Completion, 2PC

This ensure automatic activation, registration, propagation and atomic termination of Web services.

WS-ATOMIC TRANSACTION

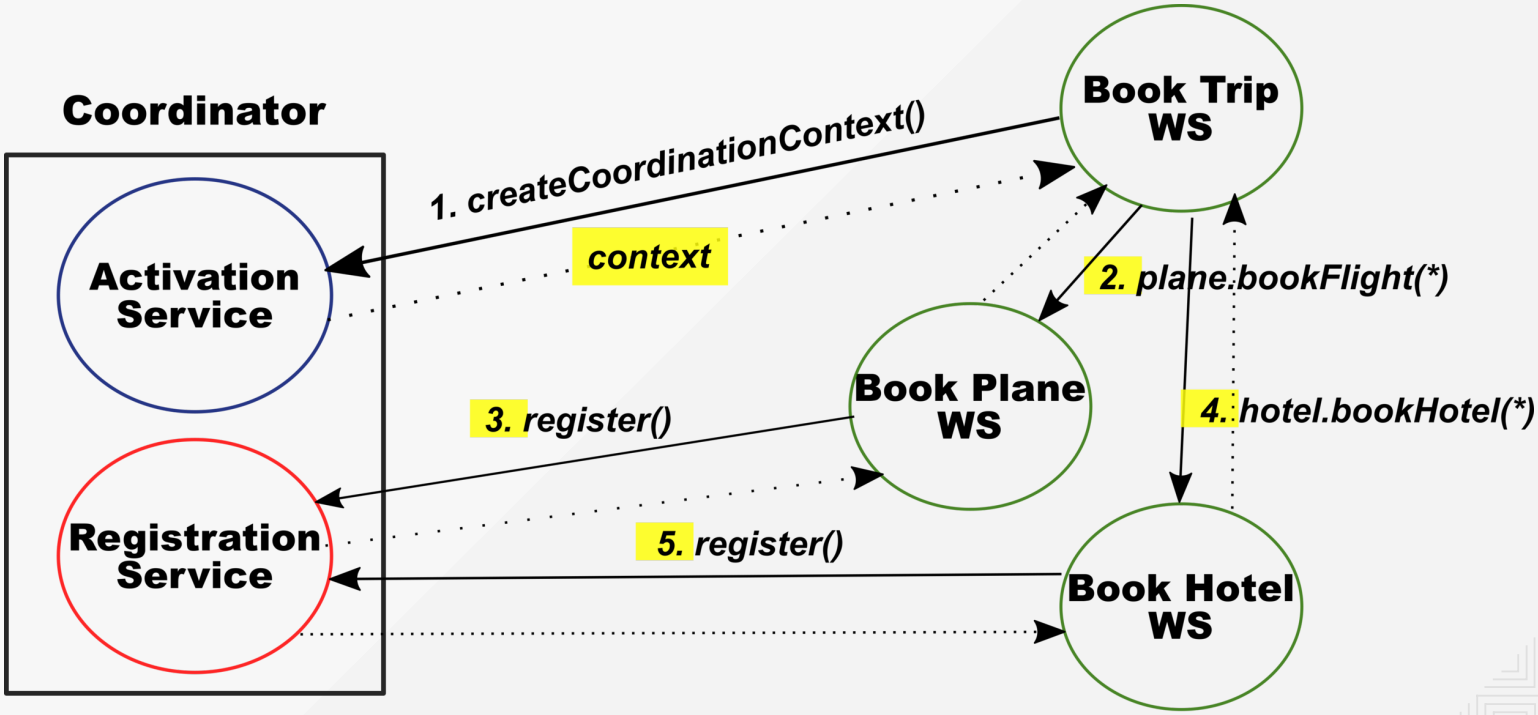
COORDINATING PROTOCOLS

There are two coordinating protocols defined to ensure atomicity.

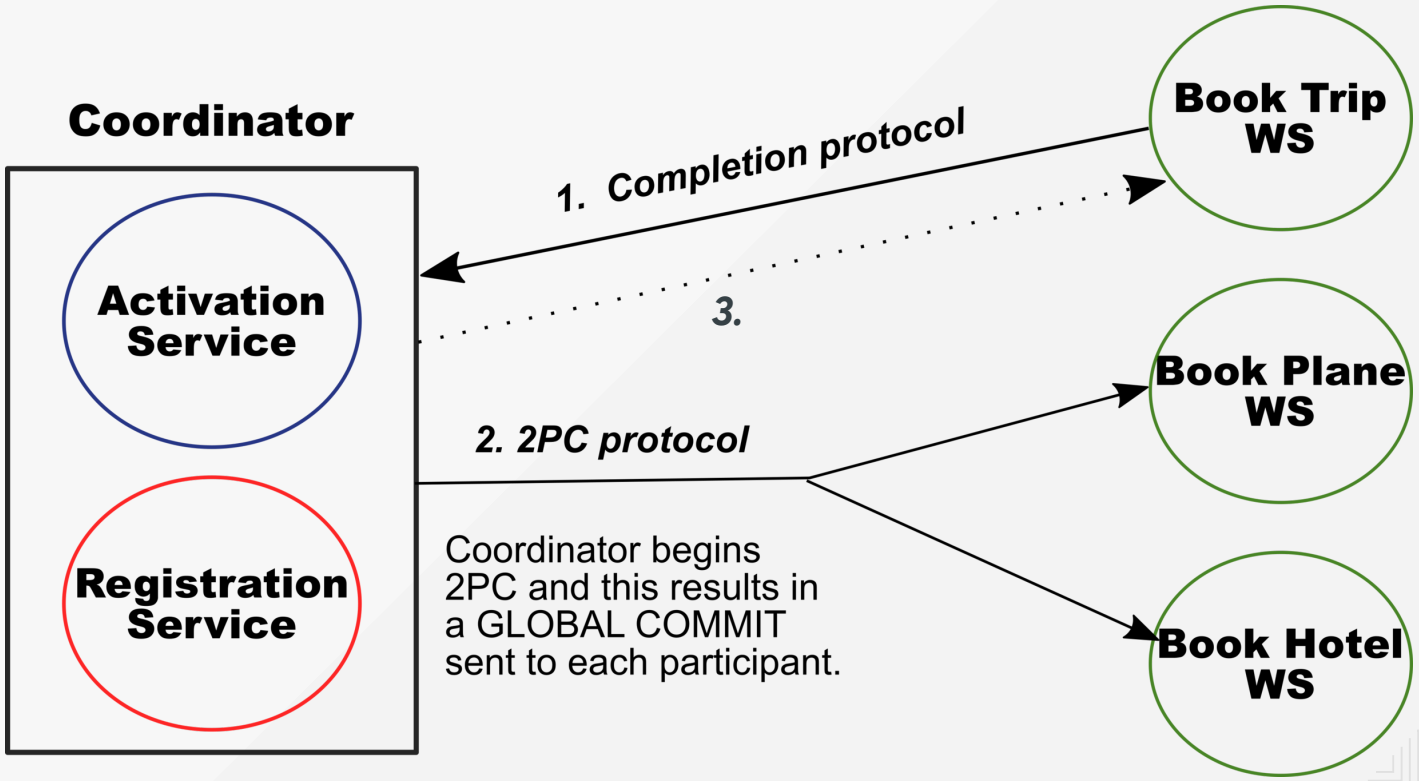
Each client/participant is registered to any of these protocols:

- **Completion** used between client and coordinator to instruct to commit or rollback
- **2PC** used between coordinator and particiant
    1. **volatile 2PC** - when commit notification is received from completion protocol, happens before durable 2PC, volatile resources e.g. caches
    2. **durable 2PC** - after successful completing of the prepare phase for Volatile 2PC participant, durable resources e.g. database

WS-ATOMIC TRANSACTION



WS-ATOMIC TRANSACTION



redhat®

THANK YOU!