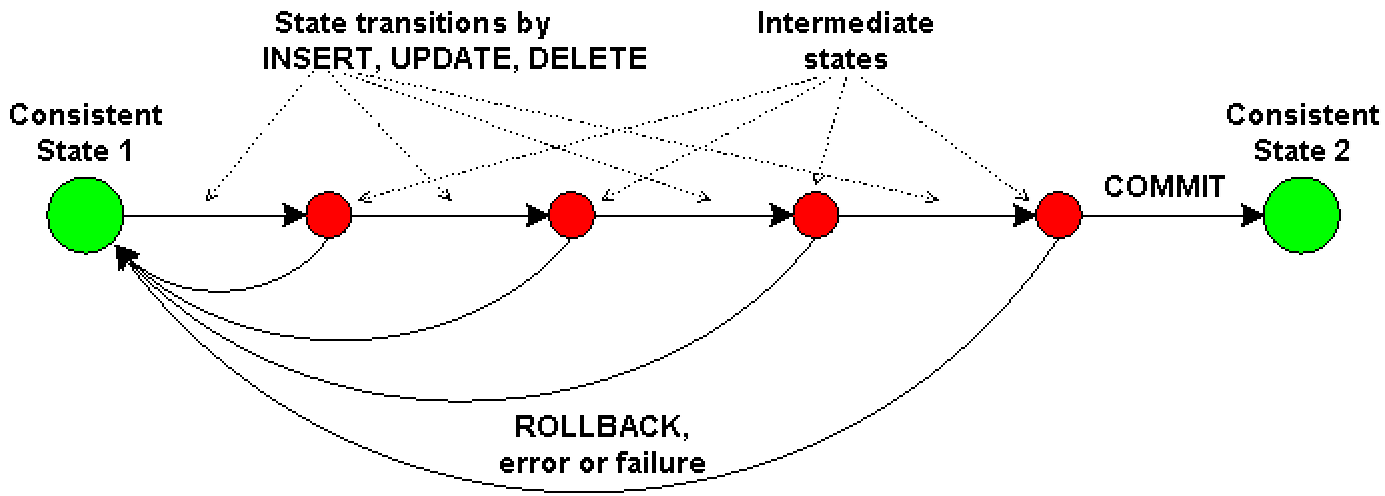
Spring Transaction Management

Table of Content

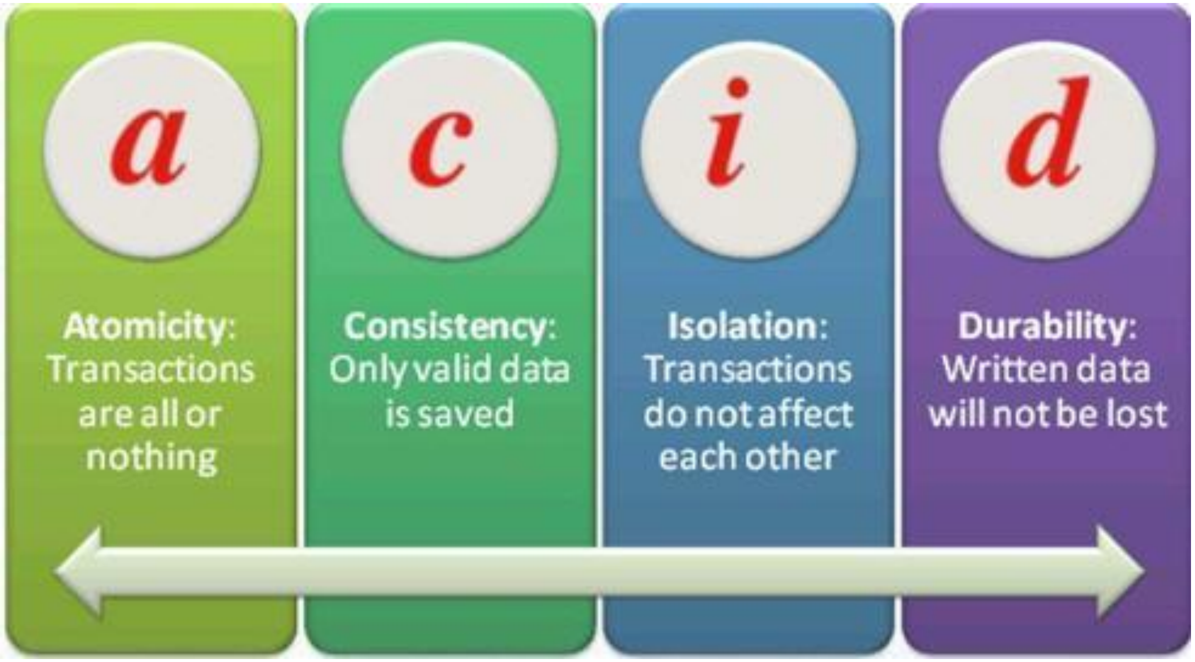
Module	Topic
Module 1	Introduction to Spring framework
Module 2	Getting Started
Module 3	Spring IoC and DI
Module 4	Bean life cycle and callback methods
Module 5	Introduction to Aspect Oriented Programming
Module 6	The AOP Advice types and point cuts
Module 7	Spring JDBC
Module 8	Spring Transaction support
Module 9	Spring support and integration

What is a Transaction?



- A transaction is a sequence of operations performed as a single logical unit of work.
- For example, transferring amount from one to other bank account, withdrawing cash at ATM center or making airline reservation etc.
- Reliability of transaction is given by ACID properties.

ACID properties



a

Atomicity: Transactions are all or nothing

c

Consistency: Only valid data is saved

i

Isolation: Transactions do not affect each other

d

Durability: Written data will not be lost

ACID properties

Atomicity

"all or nothing": if one part of the transaction fails, the entire transaction fails, and the database state is left unchanged.

Consistency

Ensures that any transaction will bring the database from one valid state to another. Any data written to the database must be valid according to all defined rules (constraints, cascades, triggers etc).

ACID properties

Isolation

Ensures that the concurrent execution of transactions results in a system state that could have been obtained if transactions are executed serially.

Durability

Means that once a transaction has been committed, it will remain so, even in the event of power loss, crashes, or errors.

Transaction Isolation Levels

If database is opened to multiple simultaneous transactions then it leads to the following problems:

- Dirty read
  - Permitted to read uncommitted or dirty data.
- Nonrepeatable read
  - Value of resource is changed while reading it multiple times within the same transaction.
- Phantom read
  - Conditional query executed multiple times within the same transaction give different result.

Transaction Isolation Levels

Isolation Level	Dirty Read	Nonrepeatable Read	Phantom Read
READ UNCOMMITTED	Allowed	Allowed	Allowed
READ COMMITTED	x	Allowed	Allowed
REPEATABLE READ	x	x	Allowed
SERIALIZABLE	x	x	x

Types of Transactions

- Flat Transactions
- Nested Transactions
- Distributed Transactions

Flat Transactions

- In a flat transaction, each transaction is decoupled from and independent of other transactions in the system.
- Another transaction cannot start in the same thread until the current transaction ends.
- For example:

Transaction 1: (Transfer amount 5000 from Tom to Jerry)

debit(Tom A; Amount 5000): Tom = Tom – 5000;

credit(Account Jerry; Amount 5000): Jerry = Jerry + 5000;

Transaction 2: (Transfer amount 10000 from Ivan to Mike)

debit(Account Ivan; Amount 10000): Ivan = Ivan – 10000;

credit(Account Mike; Amount 10000): Mike = Mike + 10000;

Nested Transactions

Nested transaction is a transaction within another transaction.

Nested transaction is a database transaction that is started by an instruction within the scope of an already started transaction.

For example:

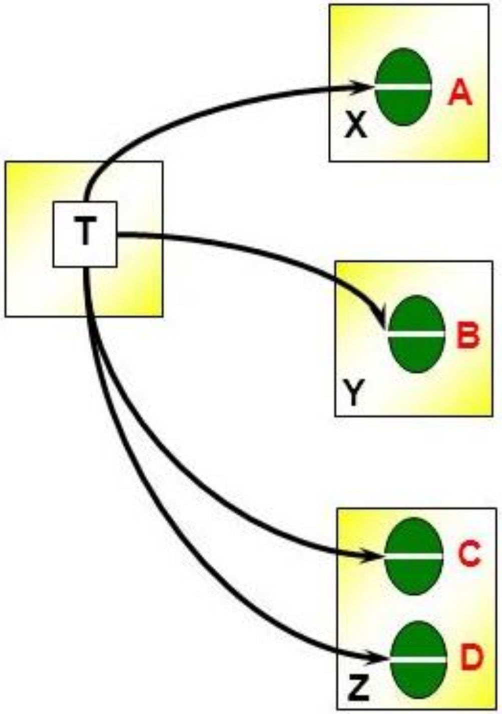
Reserve flight : Mumbai to Paris

Reserve flight : Paris to New York (Nested Transaction)

Nested Transactions Rules

- While the nested (child) transaction is active, the parent transaction may not perform any operations other than to commit or abort, or to create more child transactions.
- Committing or aborting a nested transaction has no effect on the state of the parent transaction. The parent transaction is still uncommitted or not aborted.
- If the parent transaction commits or aborts while it has active children, the child transactions are resolved in the same way as the parent.

Distributed Transactions



- Distributed transaction is a database transaction in which two or more network hosts are involved.
- Distributed transactions satisfies ACID properties.

Java technologies supporting transactions

- JDBC
- Java Persistence API (JPA)
- Java Data Objects (JDO)
- Java Transaction API (JTA)
- ORM (Hibernate, TopLink, iBatis etc.)
- EJB (Enterprise Java Beans)

JDBC

JDBC is a API interface that establishes communication between Java application & database.

```
Connection dbcon = dataSource.getConnection();
Statement stmt = dbcon.createStatement();
try {
        dbcon.setAutoCommit(false);
        stmt.executeUpdate(SQL_1);
        stmt.executeUpdate(SQL_2);
        dbcon.commit();
}catch(SQLException e) {
        dbcon.rollback();
}
```

Java Persistence API (JPA)

The JPA is a Java application programming interface specification that describes the management of relational data in applications using J2SE & J2EE.

```
Employee employee = new Employee();
employee.setFirstname(“John");
employee.setId(1);

EntityManager em = factory.createEntityManager();

em.getTransaction().begin();
em.persist(employee);
em.getTransaction().commit();
```

Java Data Objects (JDO)

JDO is a standard way to access persistent data in databases, using plain old Java objects (POJO) to represent persistent data.

```
PersistenceManager pm = pmf.getPersistenceManager();
Transaction tx = pm.currentTransaction();
try {
    tx.begin();
    {users code to persist objects}
    tx.commit();
} catch(Exception e {
        tx.rollback();
}
```

Java Transaction API (JTA)

JTA is an API provided by J2EE that is used manage transactions in enterprise applications.

```
UserTransaction ut =
(UserTransaction)jndiCtx.lookup("javax.transaction.UserTransaction");

try {
        ut.begin();
        //fire SQL on DB.
        ut.commit();
}
catch(Exception e) {
        ut.rollback();
}
```

Object Relational Mapping (ORM)

ORM is a technique that converts objects into relational data & vice versa. There are several ORMs provided on Java platform like hibernate, iBatis, toplink etc.

```
Transaction tx = null;
try{
        Session session = HibernateUtil.getSessionFactory().openSession();
        tx = session.beginTransaction();
        //doSomething(session);
        tx.commit();

}catch(RuntimeException e){
        tx.rollback();
}
```

EJB (Enterprise Java Beans)

EJB offers extensive support for transaction management. In EJB we can achieve transaction support in 2 ways:

- Container Managed Transactions (CMT)
- Bean Managed Transactions (BMT)

CMT or Declarative Transactions

```
@TransactionManagement(TransactionManagementType.CONTAINER)
public class AccountBean implements AccountRemote {

@TransactionAttribute(TransactionAttributeType.REQUIRED)
   public void transferFunds() {
      //fund transfer logic
   }
}
```

BMT or Programmatic Transactions

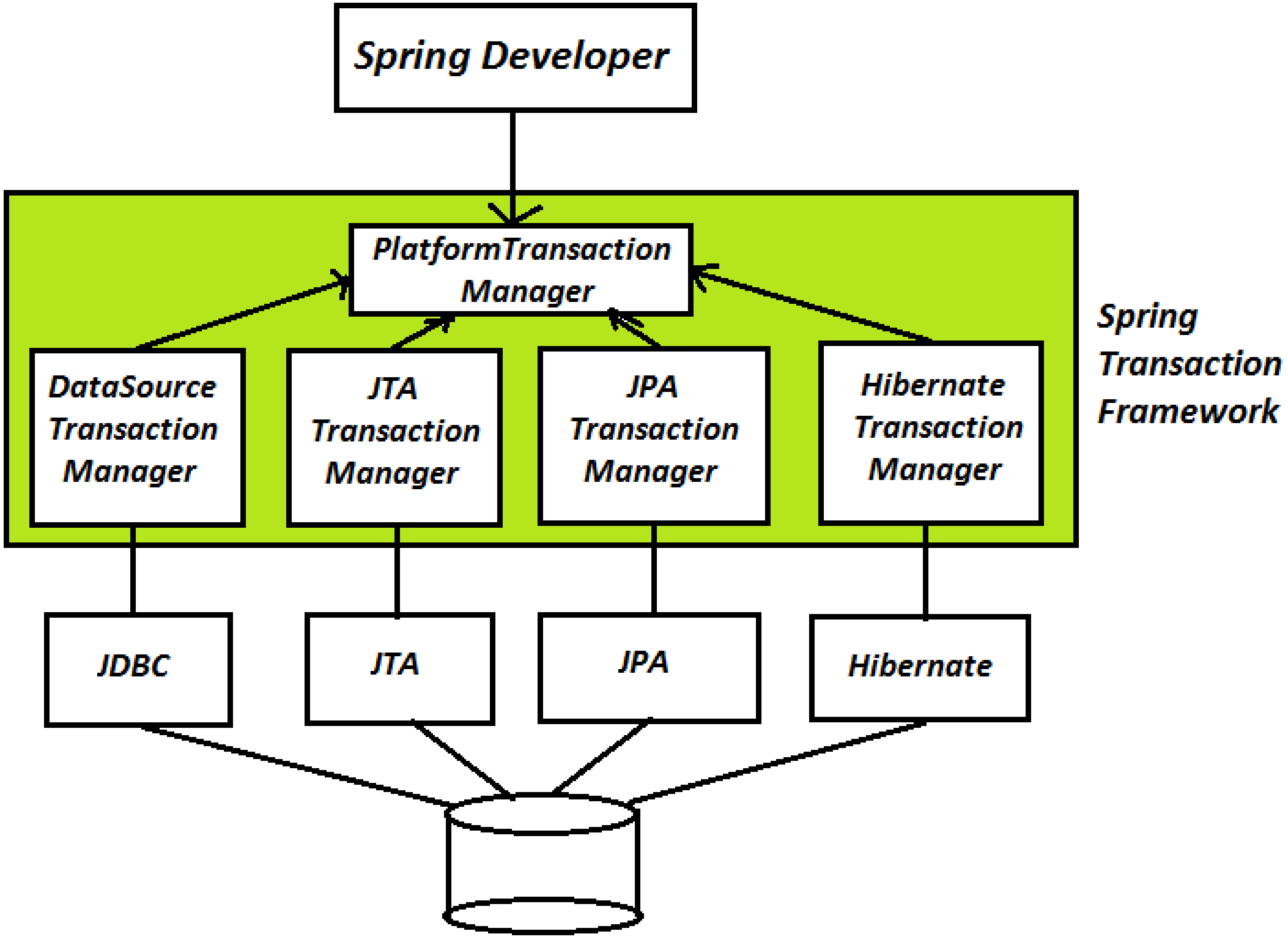
```
@TransactionManagement(TransactionManagementType.BEAN)
public class AccountBean implements AccountRemote {
        @Resource
        UserTransaction userTransaction;

public void transferFunds(acc1, acc2, amount) {
         userTransaction.begin();
             //fund transfer logic
         userTransaction.commit();
  }
}
```

Problems with EJB transactions

- To use CMT, requires application server.
- BMT is possible using only JTA.

Spring Transaction Framework


Spring Developer
PlatformTransaction Manager
DataSource Transaction Manager
JTA Transaction Manager
JPA Transaction Manager
Hibernate Transaction Manager
Spring Transaction Framework
JDBC
JTA
JPA
Hibernate

Advantages of spring transactions

- Spring offers both programmatic & declarative transactions.
- Declarative transactions are equivalent to CMT & hence no need to have an application server.
- You may not use JTA always.
- Spring transaction offers variety of transaction managers those give freedom to developers while choosing transaction API.

Transaction manager configuration

```
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource" />
</bean>

<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">
        <property name=“transactionManagerName" ref=“java:/TransactionManager" />
</bean>

<bean id="transactionManager"
class="org.springframework.orm.jpa.JpaTransactionManager">
        <property name=“entityManagerFactory" ref="entityManagerFactory" />
</bean>

<bean id="transactionManager"
class="org.springframework.orm.hibernate.HibernateTransactionManager">
        <property name=“sessionFactory" ref="sessionFactory " />
</bean>
```

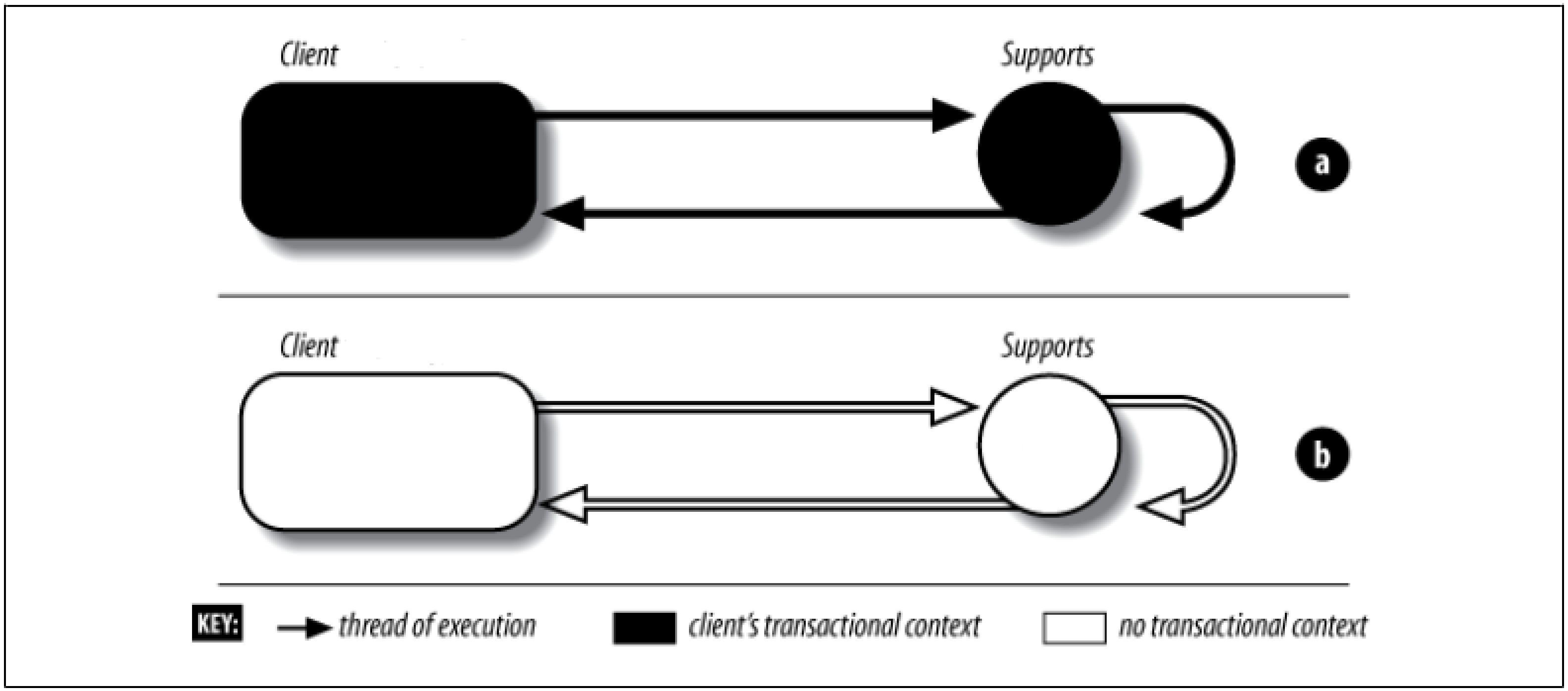
Programmatic Transactions

- In programmatic transactions, developer is responsible to decide when to begin & when to commit the transaction.
- Programmatic transactions give a greater control to the developer in setting up the transaction boundaries.

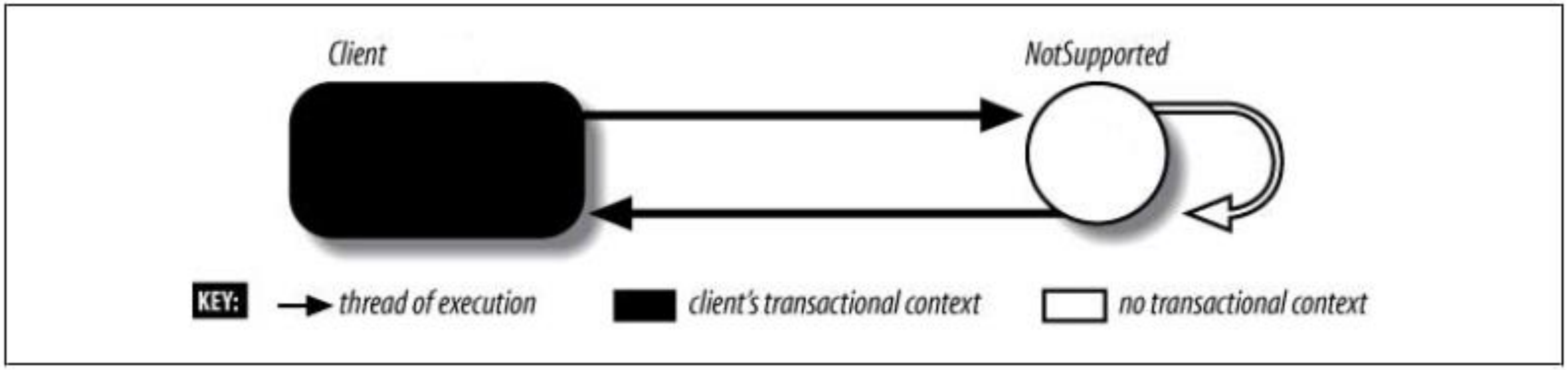
Transaction Attributes

- Supports
- Not Supported
- Required
- RequiresNew
- Never
- Mandatory

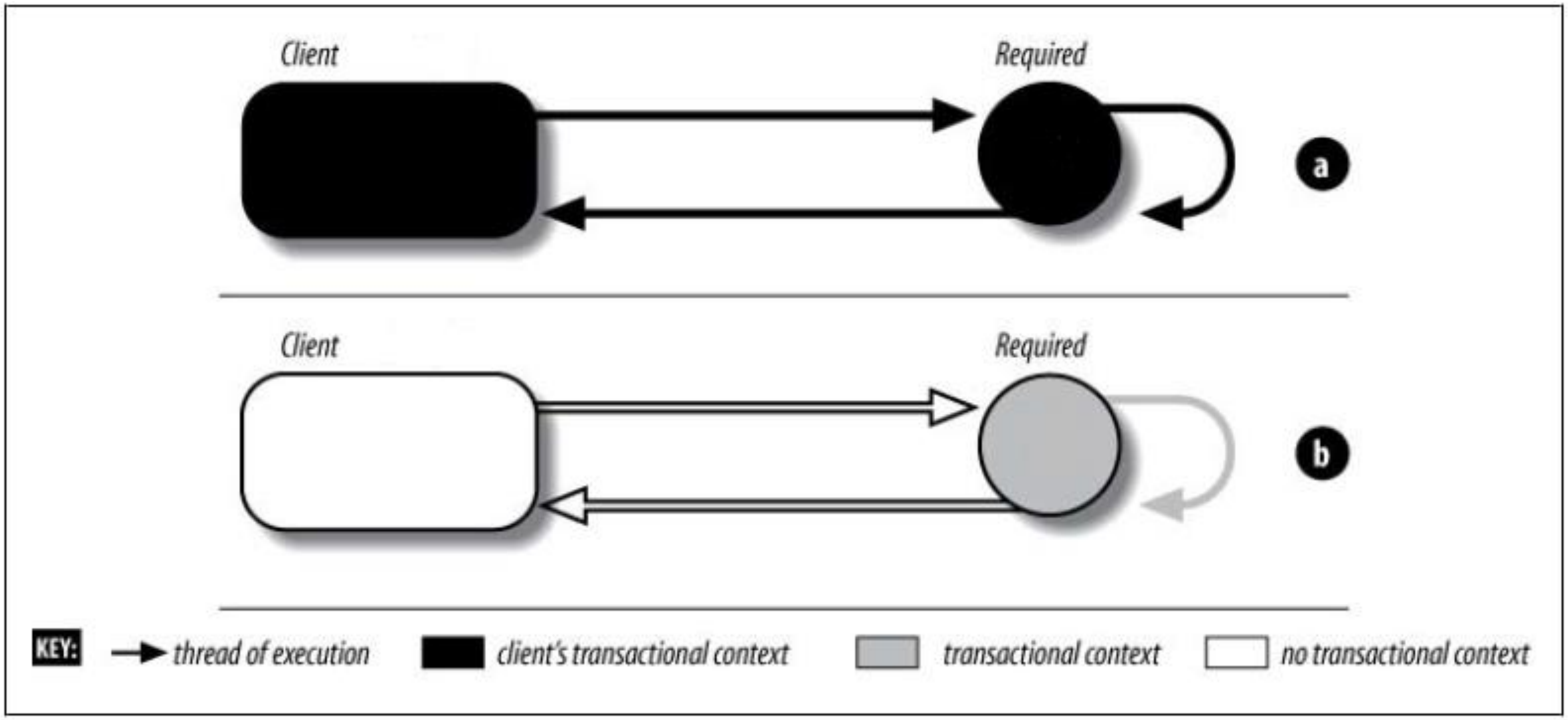
Supports


Client
Supports
a
Client
Supports
b
KEY:
thread of execution
client's transactional context
no transactional context

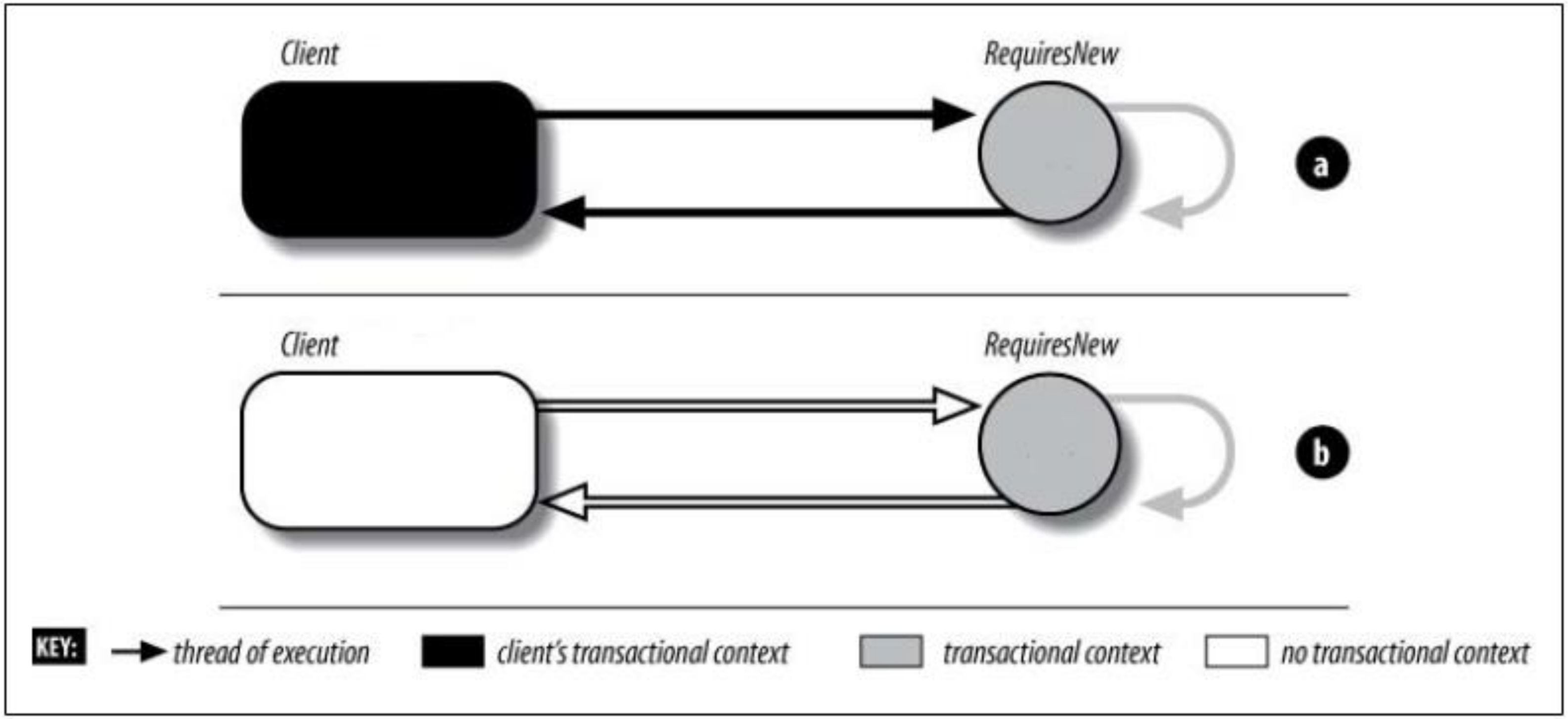
Not Supported



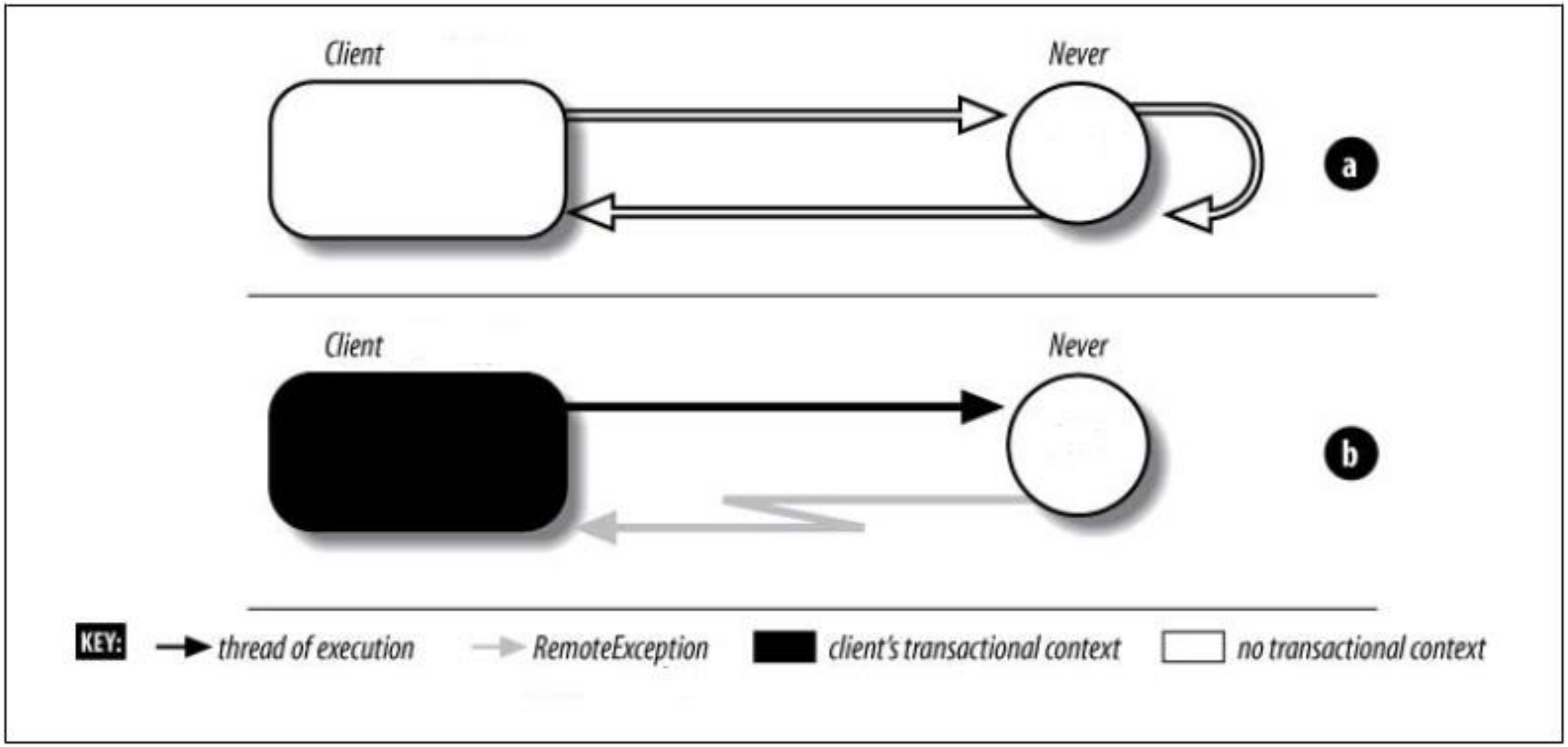
Required


Client
Required
a
Client
Required
b
KEY:
thread of execution
client's transactional context
transactional context
no transactional context

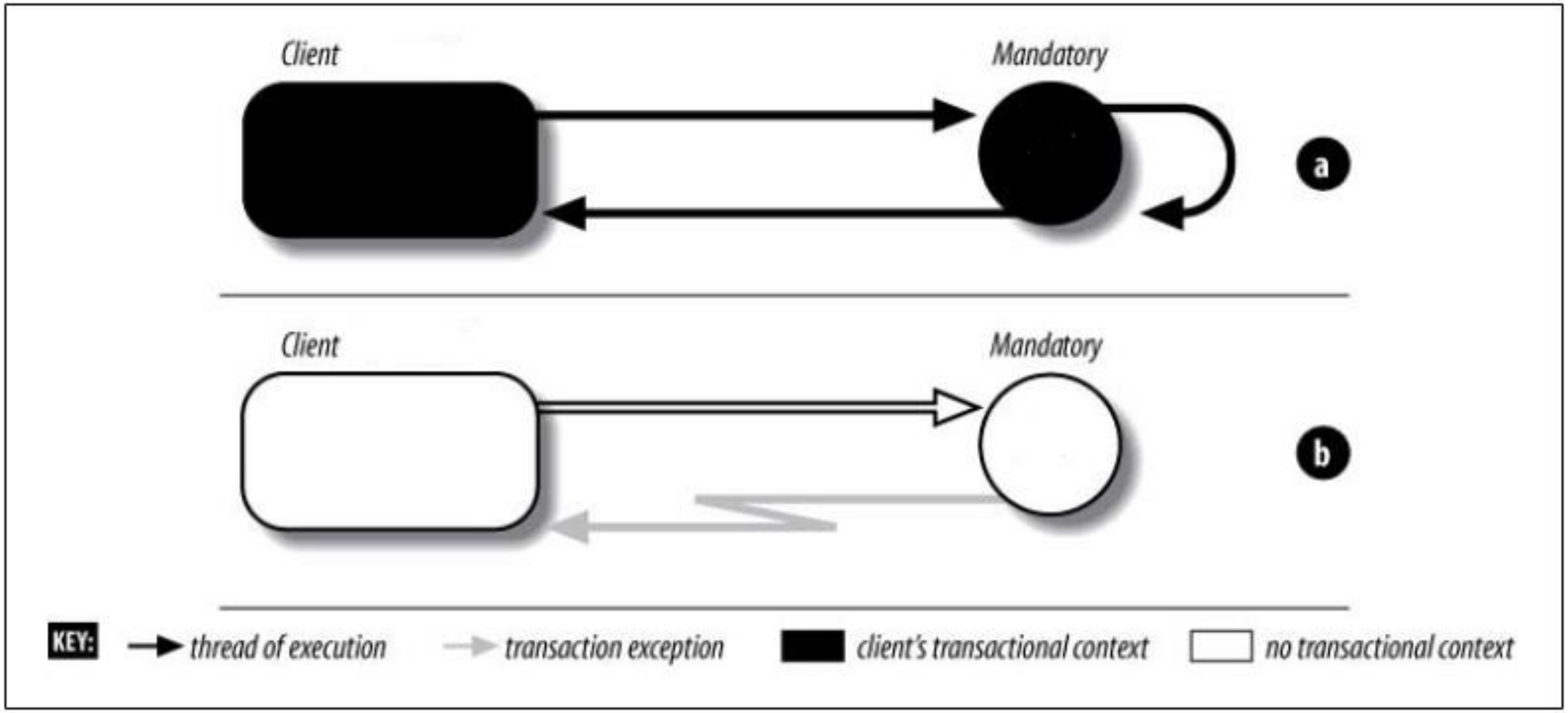
Requires New



Never


Client
Never
a
Client
Never
b
KEY:
thread of execution
RemoteException
client's transactional context
no transactional context

Mandatory


Client
Mandatory
a
Client
Mandatory
b
KEY:
thread of execution
transaction exception
client's transactional context
no transactional context