Data and Transaction Management

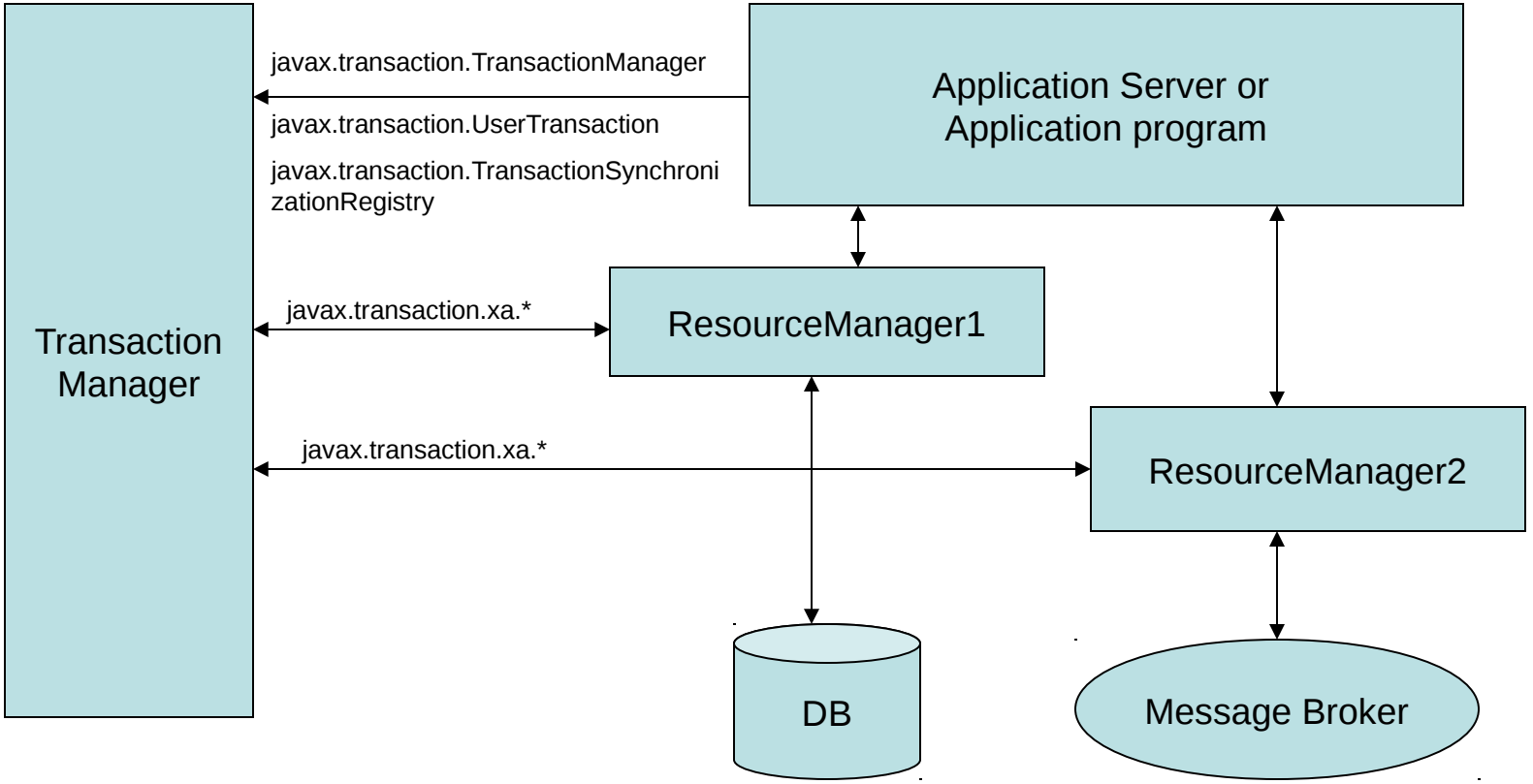
Introduction

- A transaction is a series of operations that execute as one, atomic operation
- Java Transaction API (JTA) is one of the Java Enterprise Edition (Java EE) APIs that enables distributed transactions to be done across multiple resources in a Java environment.

Transaction types

- Local transaction - A transaction that involves only one resource manager (e. g. accesses only one database). Support all the ACID properties (atomicity, consistency, isolation and durability)
- Distributed transaction - accesses and updates data on two or more networked resources (e. g. RDBMSs). Support all the ACID properties.
- Short-living transaction – support all the ACID properties.
- Long-living transaction – do not support ACID properties.

Conceptual View of DTP model



JTA API

- JTA API is modeled on the X/Open XA standard
- The key packages under this API used to facilitate transactions include:
  - javax.transaction.TransactionManager – used by the application server itself to control transactions.
  - javax.transaction.UserTransaction – provides the application the ability to control transaction boundaries programmatically.
  - Javax.transaction.TransactionSynchronizationRegistry – introduced in JSR 907. Used from frameworks for association of arbitrary objects with transactions.

javax.transaction.xa.XAResource

- start(xid, flags) - Starts work on behalf of a transaction branch specified in xid.
- end(xid, flags) - Ends the work performed on behalf of a transaction branch.
- prepare**(**xid) - Ask the resource manager to prepare the transaction specified in xid for commit.
- recover – obtain a list of prepared transaction branches which were not commited or rolledback
- commit(xid, boolean onePhase)- Informs the resource manager to commits the global transaction specified by xid.
- rollback(xid) – Informs the resource manager to rollbacks the global transaction specified by xid.

javax.transaction.xa.XID interface

XID = key for one transaction branch.

XID contains:

- Format ID – int. Must be >0, usually format id is 0
- Global transaction id - ID of the distributed transaction. This is byte[] with length up to 64 bytes.
- Branch Qualifier – ID of the transaction branch. This is byte[] with length up to 64 bytes.

Global transaction Id and branch qualifier taken together must be globally unique.

Motivations

- Atomic operations
  - Group of operation that must all succeed or all fail
- Network or machine failure
  - What happens if the network or database fails in the middle of an account transfer operation?
- Multiple users share data
  - Multiple users might be trying to modify data at the same time and could create inconsistencies

Properties of Transactions

- Transactions' ACID properties mark the main GOALS of Transactions:
  - Atomicity
  - Consistency
  - Isolation
  - Durability

Atomicity

- Transactions are committed or rolled back as a group, and are atomic, meaning that all SQL statements contained in a transaction are considered to be a single indivisible unit.

Consistency

- Transactions ensure that the database state remains consistent, meaning that the database starts at one consistent state and ends in another consistent state when the transaction finishes.

Isolation

- Separate transactions should appear to run without interfering with each other.

Durability

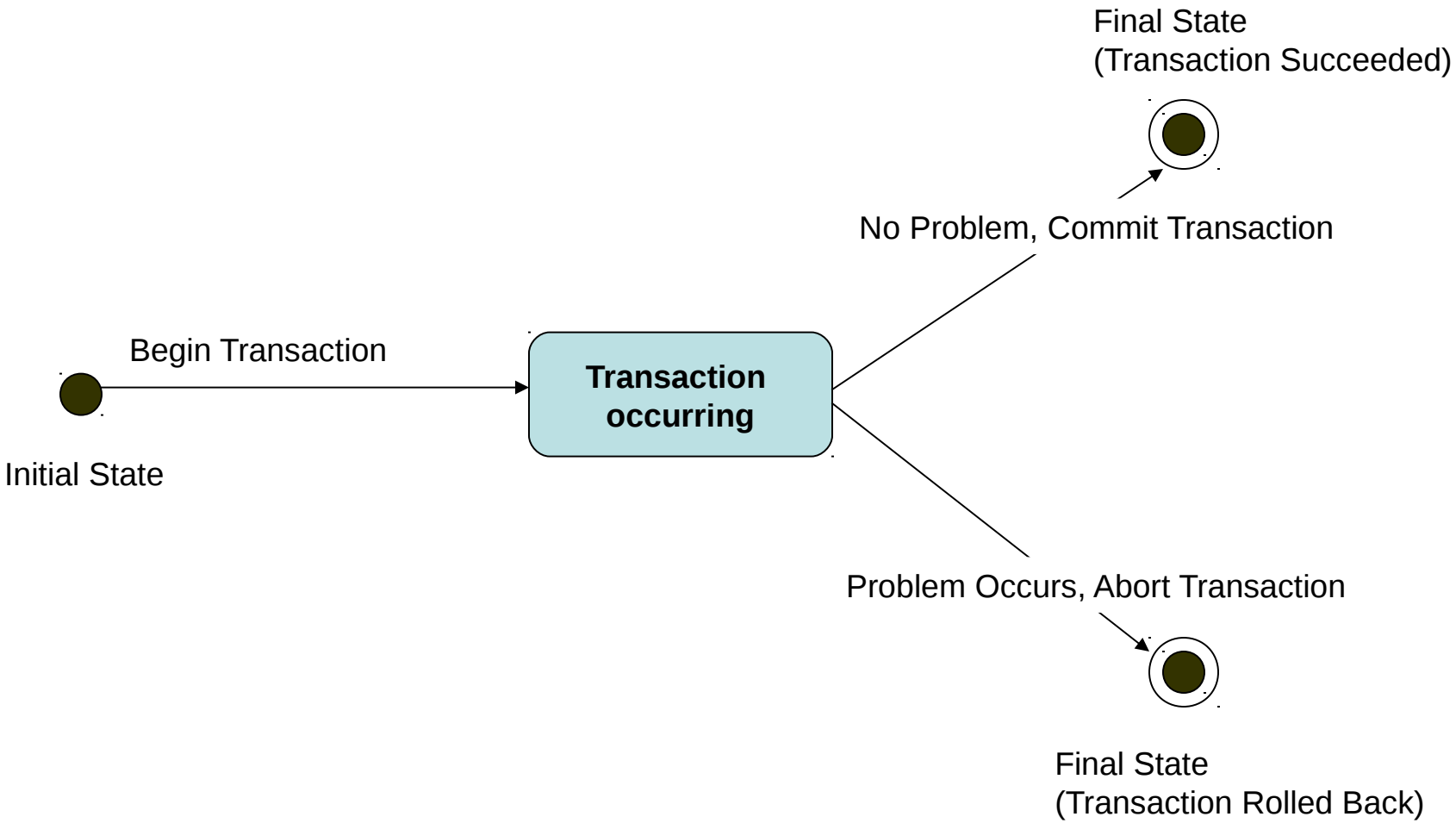
- Once a transaction has been committed, the database changes are preserved, even if the machine on which the database software runs later crashes.

The next slides will discuss various transaction types, alongside some other related features/environments in which transactions thrive.

Flat Transactions

- Series of operations performed as an atomic unit of work
- A successful transaction is committed, all persistent operation become permanent
- A failed transaction is aborted, none of the resource update becomes durable, all changes are rolled back
- Application can be notified of an abort so that in memory changes can be undone

Flat Transactions



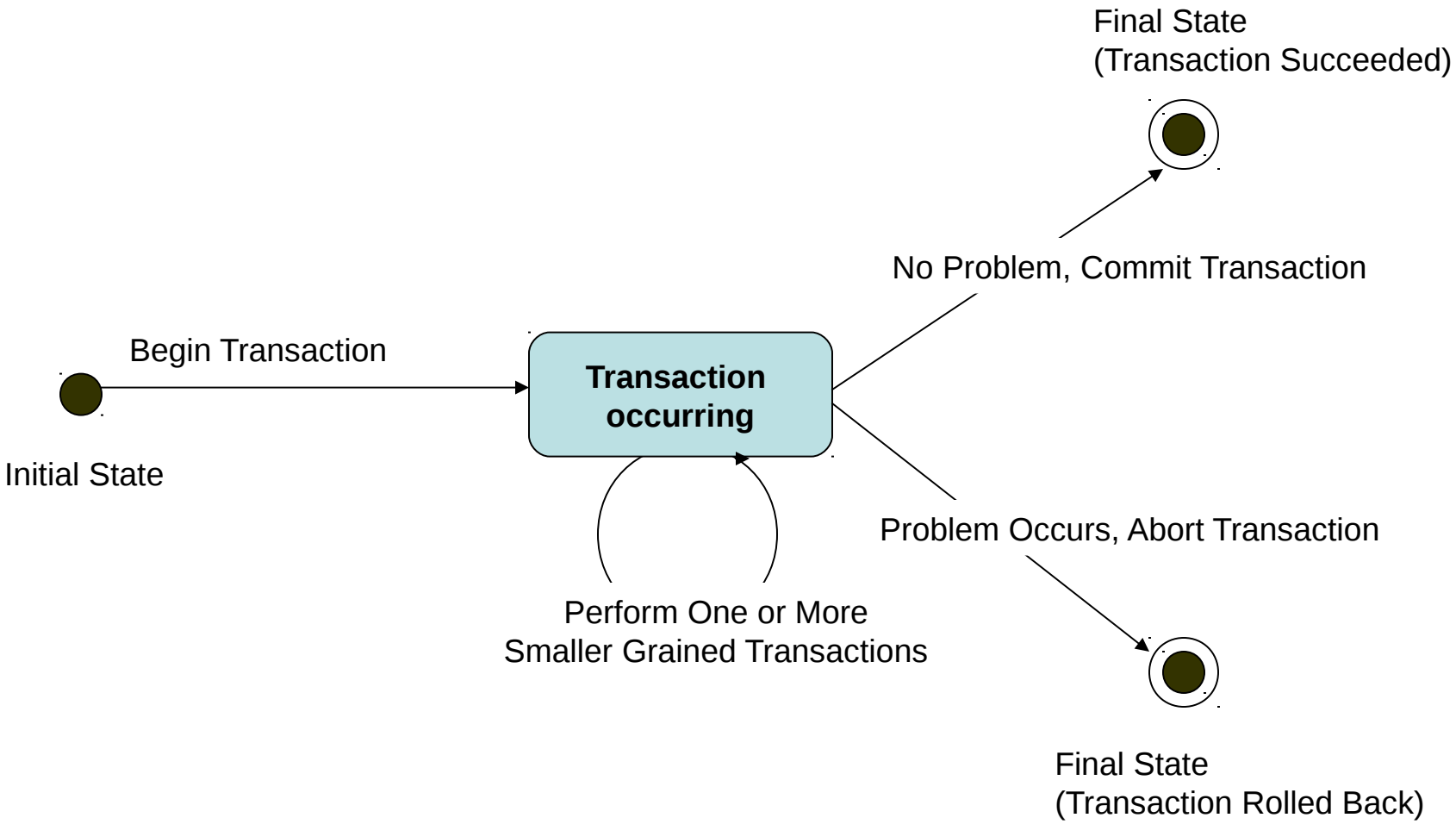
Transaction Rollback

- Transaction may abort for a number of reasons
  - Invalid parameters
  - System state violated
  - Hardware or software failure
- Rollback is obtained by forcing a commit on the underlying database only at the end of all the operations that are part of the transaction

Nested Transactions

- Embed one unit of work within another
- The nested unit of work can be rolled back without forcing the whole transaction to roll back
- The nested unit can be retried but if it does never succeed it will eventually case the whole transaction to fail
- Can be visualized as a *tree* of transactions

Nested Transactions



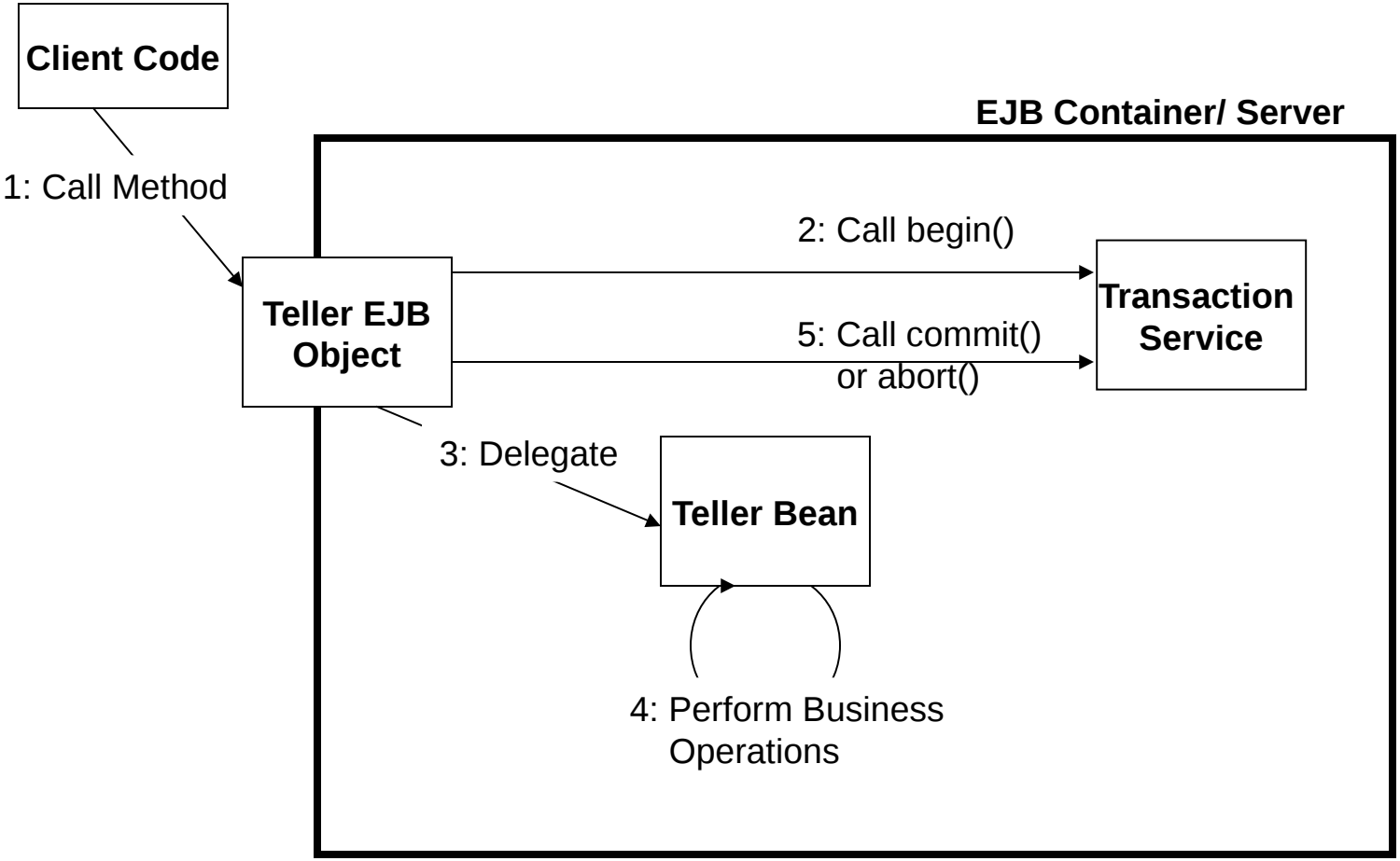
Transactions and EJB

- EJB specification only mandates support for flat transactions
- You never write explicit code to handle transactions
- Transactions are abstracted out by the EJB container and run behind the scenes

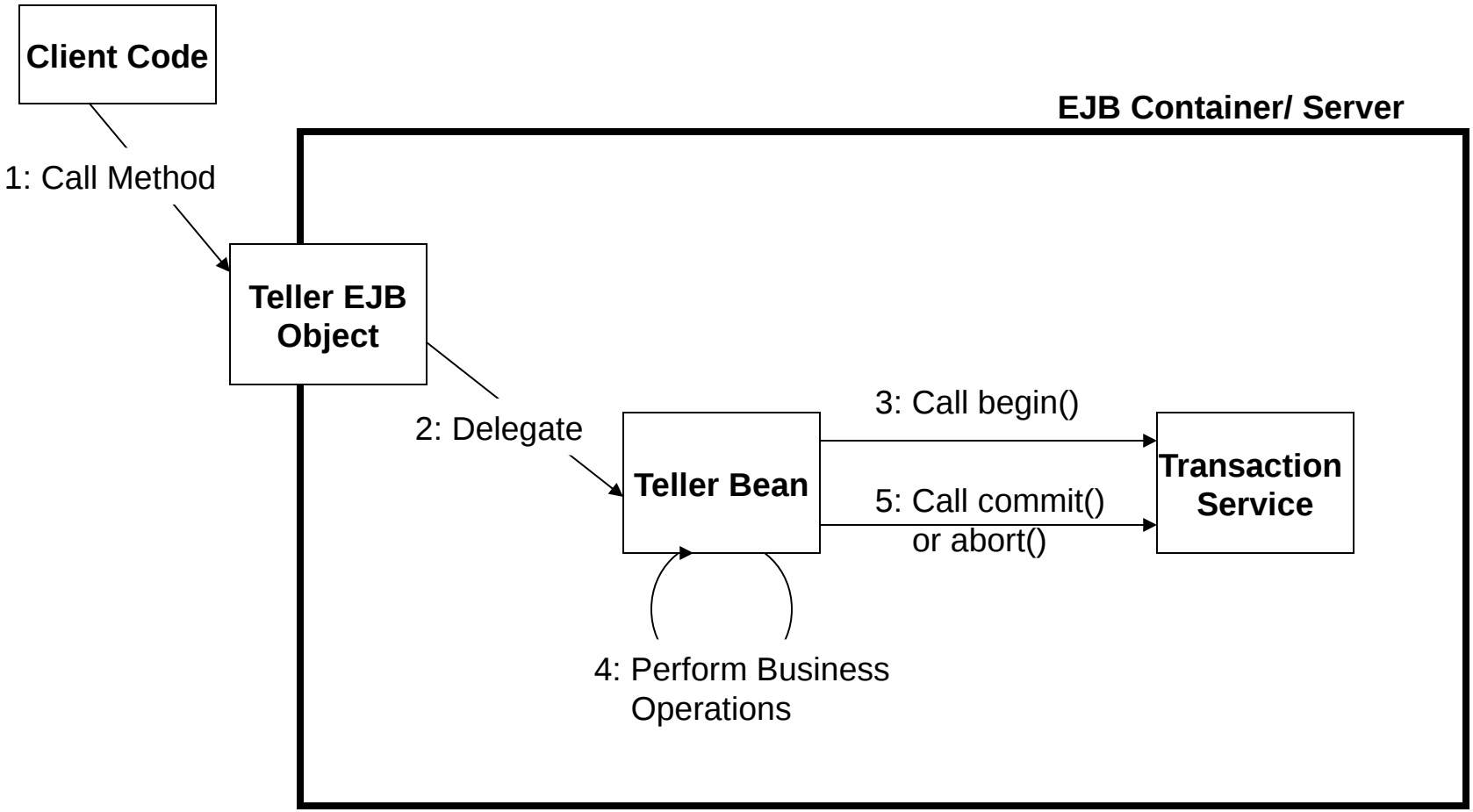
Transactional Boundaries

- Demarcating transactional boundaries means:
  - Who begins a transaction
  - Who issues a commit or abort
  - When these steps occur
- One can demarcate transactional boundaries using three key ways:
  - Programmatically
  - Declaratively
  - User-initiated

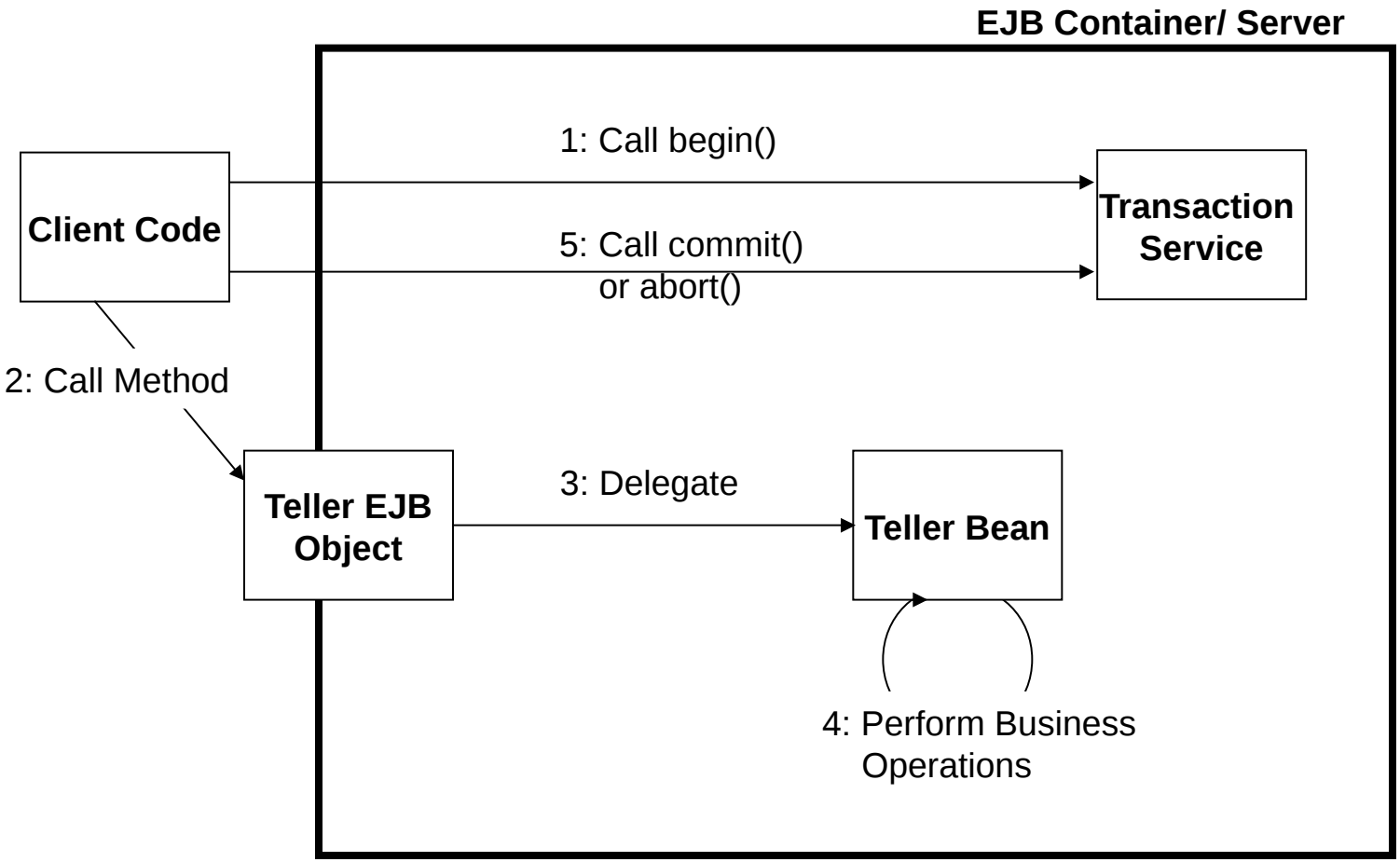
Declarative Transactions



Programmatic Transactions



Client-initiated Transactions



Container-managed Transactions

- The EJB container sets the boundaries of the transactions.
- The bean code does not include statements that begin and end the transaction.
- Can be used with any type of enterprise bean
- Typically, the container:
  - begins a transaction immediately before an enterprise bean method starts.
  - commits the transaction just before the method exits.
- Each method can be associated with a single transaction
- When deploying a bean, you specify which of the bean's methods are associated with transactions by setting transaction attributes.

Transaction Attributes

- Required
  - If the client is running within a transaction and invokes the enterprise bean's method, the method executes within the client's transaction.
  - If the client is not associated with a transaction, the container starts a new transaction before running the method.
- RequiresNew
  - Always create a new transaction when the bean is called
- NotSupported
  - The bean cannot be involved in a transaction at all
  - Used if you are sure you do not need the ACID properties

Transaction Attributes

- Supports
- Runs in a transaction only if the client has one running already
- Mandatory
  - Mandates that a transaction be already ruinning when the bean is called
  - If a transaction is not running `javax.ejb.TransactionRequiredException` is raised
- Never
  - The bean cannot be involved in a transaction
  - If called within a transaction will raise an exception (`java.rmi.RemoteException` or `Javax.ejb.EJBException`

Transaction Attributes

Transaction Attribute	Client's Transaction	Bean's Transaction
Required	None	T2
	T1	T1
RequiresNew	None	T2
	T1	T2
Mandatory	None	Error
	T1	T1
NotSupported	None	None
	T1	None
Supports	None	None
	T1	T1
Never	None	None
	T1	Error

Transaction Attributes

Transaction Attribute	Stateless Session Bean	Stateful Session Bean	Entity Bean	Message-driven Bean
Required	Yes	Yes	Yes	Yes
RequiresNew	Yes	Yes	Yes	No
Mandatory	Yes	Yes	Yes	No
Support	Yes	No	No	No
NotSupported	Yes	No	No	Yes
Never	Yes	No	No	No

javax.transaction.UserTransaction

```
public interface javax.transaction.UserTransaction{
  public void begin();
  public void commit();
  public int getStatus();
  public void rollback();
  public void setRollbackOnly();
  public void setTransactionTimeout(int);
}
```

javax.transaction.Status

```
public interface javax.transaction.Status{
  public static final int STATUS_ACTIVE;
  public static final int STATUS_NO_TRANSACTION;
  public static final int STATUS_MARKED_ROLLBACK;
  public static final int STATUS_PREPARING;
  public static final int STATUS_PREPARED;
  public static final int STATUS_COMMITTING;
  public static final int STATUS_COMMITTED;
  public static final int STATUS_ROLLING_BACK;
  public static final int STATUS_ROLLEDBACK;
  public static final int STATUS_UNKNOWN;
}
```