

CORBA part II

Common CORBA Services

CCS

CORBA Services

- A set of interface specifications provides fundamental services that application developers may need in order to
 - Find objects
 - Manage objects
 - Coordinate the execution of complex operations
- CORBA Services are the building blocks for other components in the OMA including applications

CORBA Services

- The following services exist:

Naming	Licensing
Notification (event)	Query
Life cycle	Properties
Persistent state	Security
Relationships	Time
Externalization	Collection
Transaction	Trading
Concurrency control	

- Some of the above are simply framework interfaces that will be inherited by the application or other objects (Ex. Life cycle service)
- Others represent low-level components on which higher-level components can be build (Ex. Transaction service)
- Others provide basic services used at all levels of applications (Ex. Naming and Trading service)

CORBA Naming Service

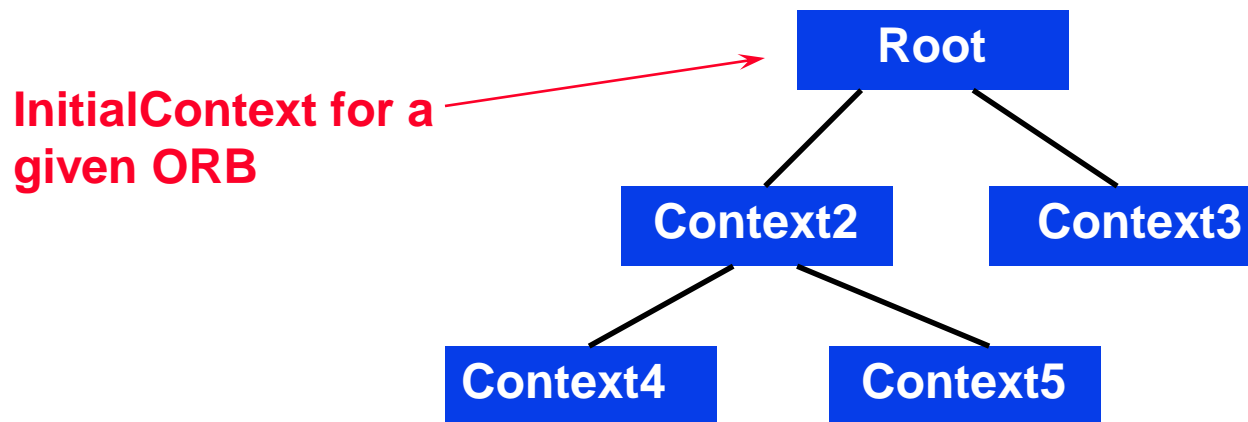
CORBA Naming Service

- Until now we used the file system and `object_to_string` to communicate the IOR to the client
- Instead we can use a Corba Naming Service to register og obtain remote references to CORBA objects in a more elegant way
- The CORBA Naming service is an example of a concrete Naming Service implementation/standard

CORBA Naming Service

CORBA Naming Service

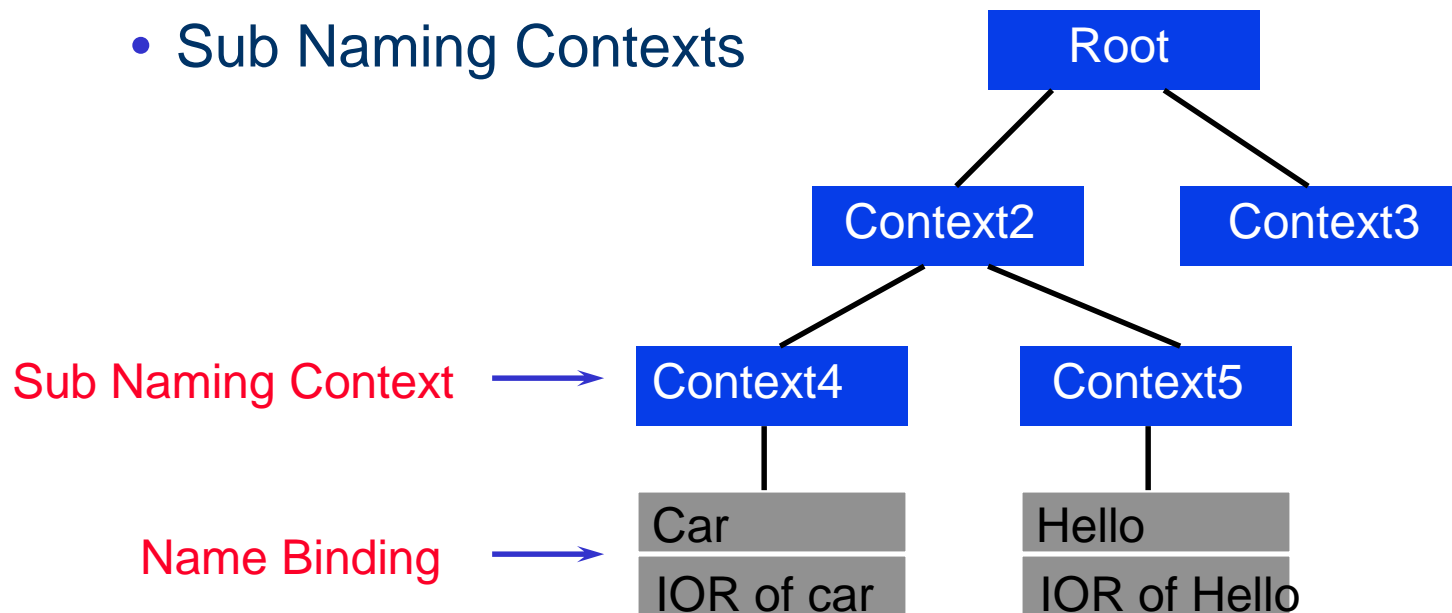
- The CORBA Naming Service is hierarchically structured in Naming Contexts
 - Naming Contexts are similar to directories in file systems



CORBA Naming Service

CORBA Naming Service

- Naming Contexts can contain
 - Name Bindings
 - Sub Naming Contexts



CORBA Naming Service

CORBA Naming Service

- Clients and Servers operates with the CORBA Naming Service in terms of Names
- A Name is a sequence of NameComponents which either can be Naming Contexts or Objects
 - Naming Contexts and Objects are identified through a string

Example: Name for an Account CORBA object

Name=

context2

context5

Hello

IDL Types for Names

```
module CosNaming {  
    typedef string Istring;  
  
    struct NameComponent {  
        Istring id;  
        Istring kind;  
    };  
  
    typedef sequence <NameComponent> Name;  
    ...  
};
```

The IDL Interfaces

- Naming Service is specified by two IDL interfaces:
 - **NamingContext** defines operations to bind objects to names and resolve name bindings.
 - **BindingIterator** defines operations to iterate over a set of names defined in a naming context.

IDL Interface

```
interface NamingContext {  
    void bind(in Name n, in Object obj)  
        raises (NotFound, ...);  
  
    Object resolve(in Name n)  
        raises (NotFound, CannotProceed, ...);  
  
    void unbind (in Name n)  
        raises (NotFound, CannotProceed...);  
  
    NamingContext new_context();  
    NamingContext bind_new_context(in Name n)  
        raises (NotFound, ...)  
    void list(in unsigned long how_many,  
        out BindingList bl,  
        out BindingIterator bi);  
};
```

IDL Interface (cont'd)

```
interface BindingIterator {  
    boolean next_one(out Binding b);  
    boolean next_n(in unsigned long how_many,  
                  out BindingList bl);  
    void destroy();  
}
```

CORBA Naming Service

CORBA Naming Service

- Name Resolution in CORBA Naming Service happens relative to a *InitialContext*
 - *resolves operation* resolves the first component of the Name, *n*, to an object reference
 - If there are no remaining Name Components, it returns this object reference to the caller
 - Otherwise it narrows the object reference to a Naming Context and passes the remainder of the Name to the *resolves()* operation
- The above algorithm will probably be optimized concrete implementations

CORBA Naming Service

CORBA Naming Service

- An application uses the Naming service this way:
 - A CORBA Server *bind* IORs to a unique Name in the Naming Server
 - The Client *resolve* to get a particular IOR by supplying a Name to the *resolve* method of a initial Naming Context
 - which you get by
`ORB.resolve_initial_references("NameService")`
 - The Client *narrow* the IOR to a given interface type and start using it

CORBA Naming Service

CORBA Naming Service – Interoperable Naming Service

- The Interoperable Naming Service (INS) provides the following features:
 - Capability to resolve using stringified names (e.g., a/b.c/d)
 - URLs for CORBA object references (corbaloc: and corbaname: formats)
 - Corbaloc -> locate CORBA services
 - Corbaname -> stringified resolvment
 - Standard APIs in NamingContextExt for converting between CosNames, URLs, and Strings
 - ORB arguments for bootstrapping (ORBInitRef and ORBDefaultInitRef)

CORBA Naming Service

CORBA Naming Service

So if I have to model a CORBA Order object in a CORBA server - will I then have to register all incarnations of orders in the Naming service?

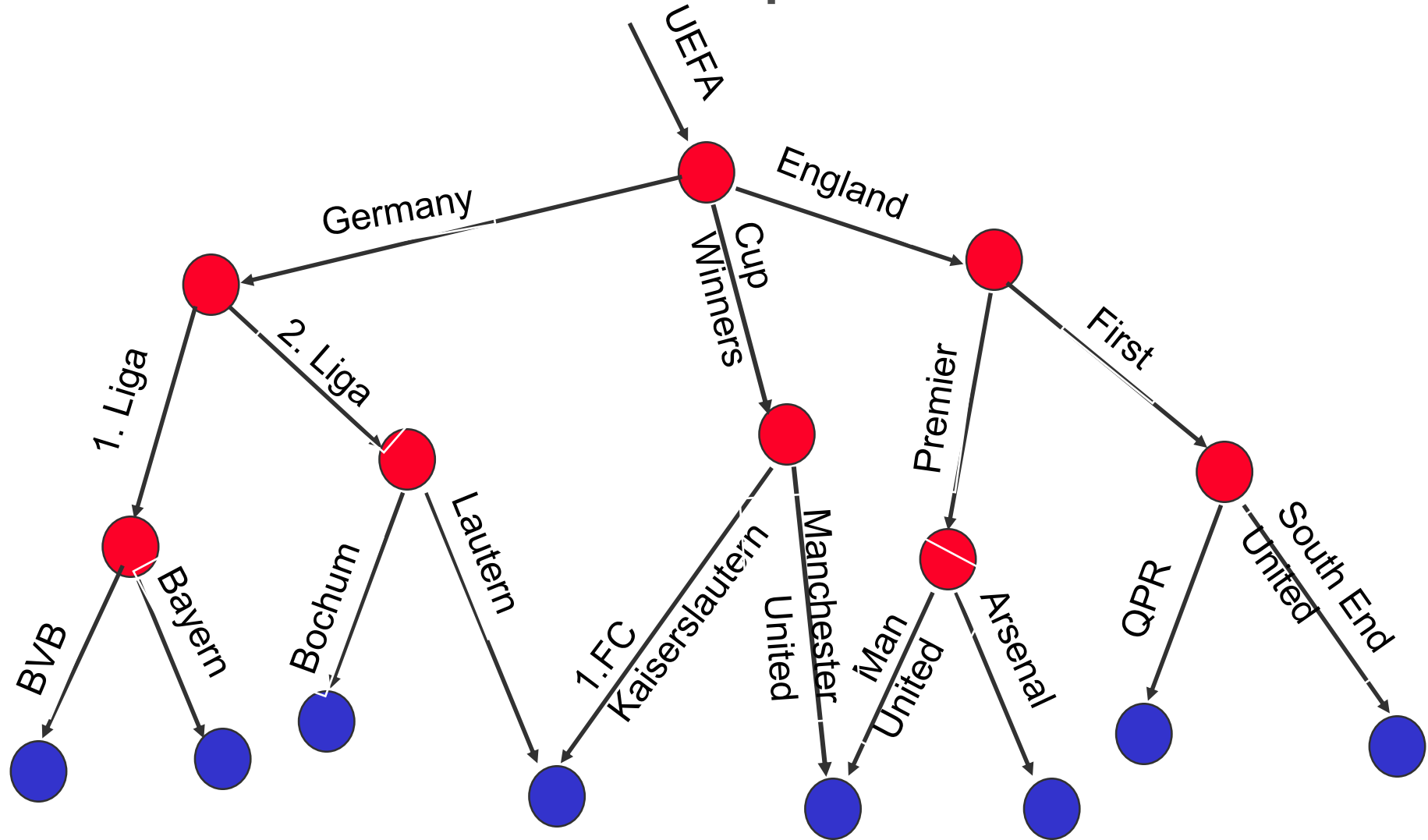


CORBA Naming Service

CORBA Naming Service

- Once again- NO!!!!
- There are more ways of comming around that problem - one common way is to apply the the Factory pattern
 - Delegate object creation to a Factory CORBA object that returns IORs
 - Bind an incarnation of the Factory object in the Naming Service
 - For clients to get IORs to specific Order objects they go through the Factory CORBA object

Naming Contexts Example



C++ Example

```
CORBA::Object *obj;
CosNaming::NamingContext *root;
Soccer::Team *t;
CosNaming::Name* name;
obj=CORBA::BOA.resolve_initial_references
    ("NameService");
root=CosNaming::NamingContext::_narrow(obj);
name=new CosNaming::Name(4);
name[0].id=CORBA::string_dupl("UEFA");
name[1].id=CORBA::string_dupl("England");
name[2].id=CORBA::string_dupl("Premier");
name[3].id=CORBA::string_dupl("Arsenal");
t=Soccer::Team::_narrow(root->resolve(name));
cout << t->print();
```

Limitations

- Limitation of Naming: Client always has to identify the server by name.
- Inappropriate if client just wants to use a service at a certain quality but does not know from who:
 - Automatic cinema ticketing,
 - Video on demand,
 - Electronic commerce.

OMG/CORBA Trading Service

Trading Characteristics

- Trader operates as broker between client and server.
- Enables client to change perspective from 'who?' to 'what?'
- Selection between multiple service providers.
- Similar ideas in:
 - yellow pages
 - insurance broker
 - stock brokerage.

Trading Characteristics

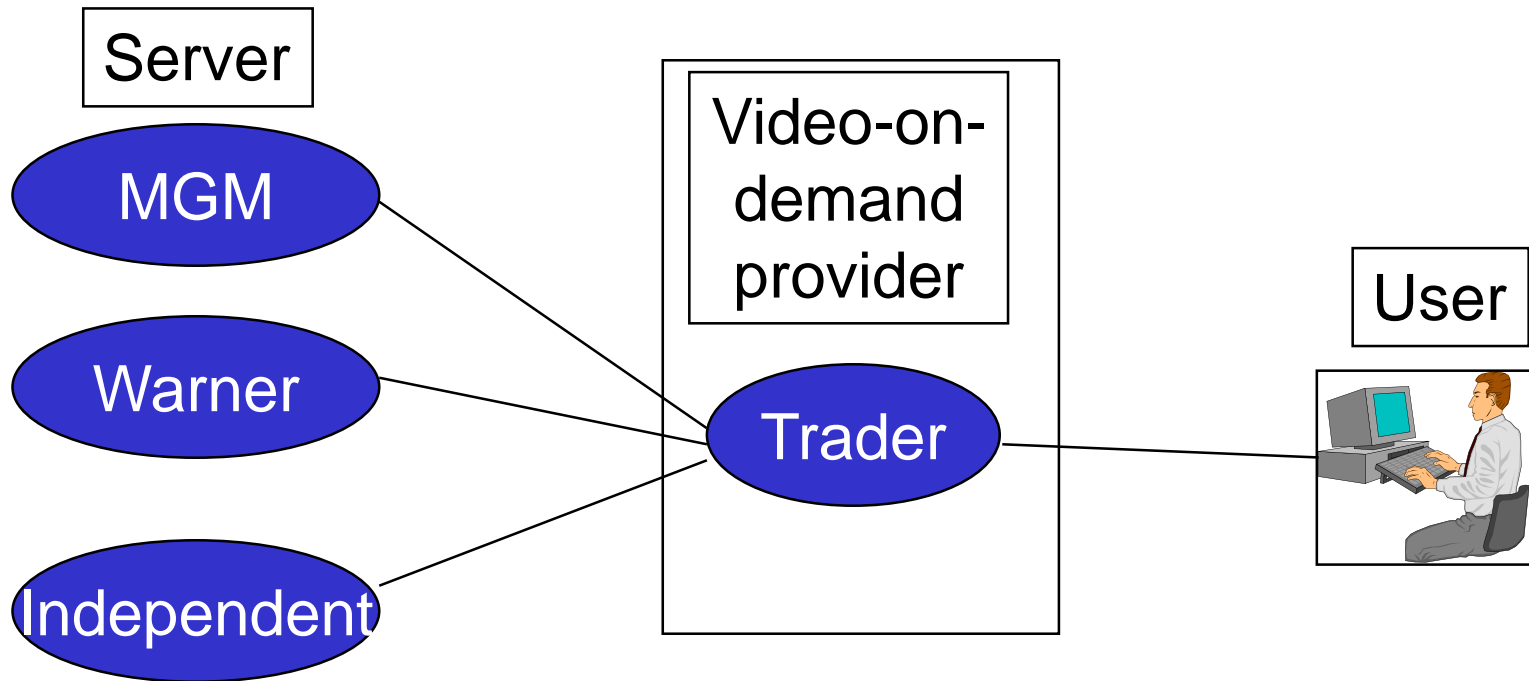
- Common language between client and server:
 - Service types
 - Qualities of service
- Server registers service with trader.
- Server defines assured quality of service:
 - Static QoS definition
 - Dynamic QoS definition.

Trading Characteristics

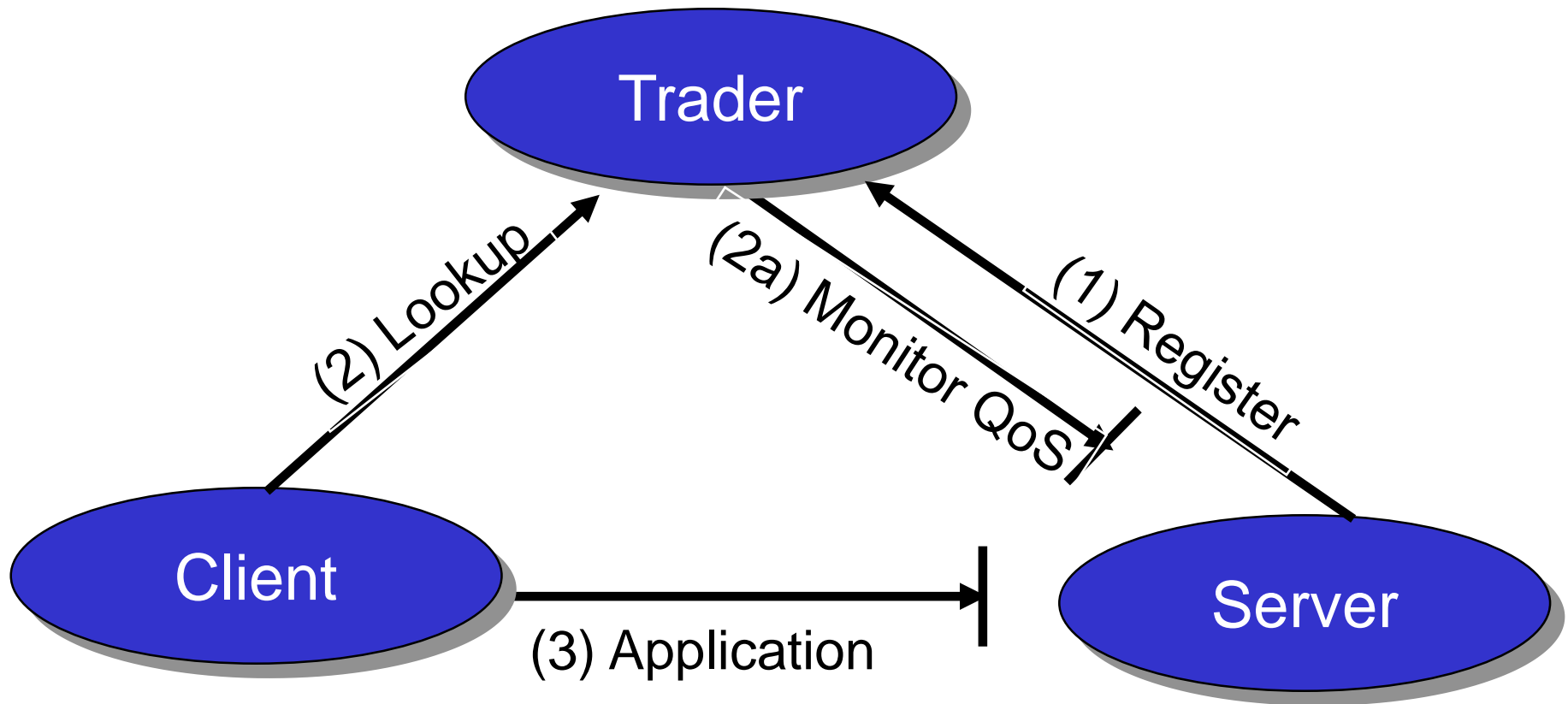
- Clients ask trader for
 - a service of a certain type
 - at a certain level of quality
- Trader supports
 - service matching
 - service shopping

Example

- Distributed system for video-on-demand:



OMG Trading Service



Properties

Specify qualities of service:

```
typedef Istring PropertyName;  
typedef sequence<PropertyName> PropertyNameSeq;  
typedef any PropertyValue;  
struct Property {  
    PropertyName name;  
    PropertyValue value;  
};  
typedef sequence<Property> PropertySeq;  
enum HowManyProps {none, some, all};  
union SpecifiedProps switch (HowManyProps) {  
    case some : PropertyNameSeq prop_names;  
};
```

Register

Trader interface for servers:

```
interface Register {  
    OfferId export(in Object reference,  
                  in ServiceTypeName type,  
                  in PropertySeq properties)  
        raises(...);  
    void withdraw(in OfferId id) raises(...);  
    void modify(in OfferId id,  
               in PropertyNameSeq del_list,  
               in PropertySeq modify_list)  
        raises (...);  
};
```

Lookup

Trader interface for clients:

```
interface Lookup {  
    void query(in ServiceTypeName type,  
               in Constraint const,  
               in Preference pref,  
               in PolicySeq policies,  
               in SpecifiedProps desired_props,  
               in unsigned long how_many,  
               out OfferSeq offers,  
               out OfferIterator offer_itr,  
               out PolicyNameSeq Limits_applied)  
    raises (...);  
};
```

Java Naming and Directory Interface JNDI

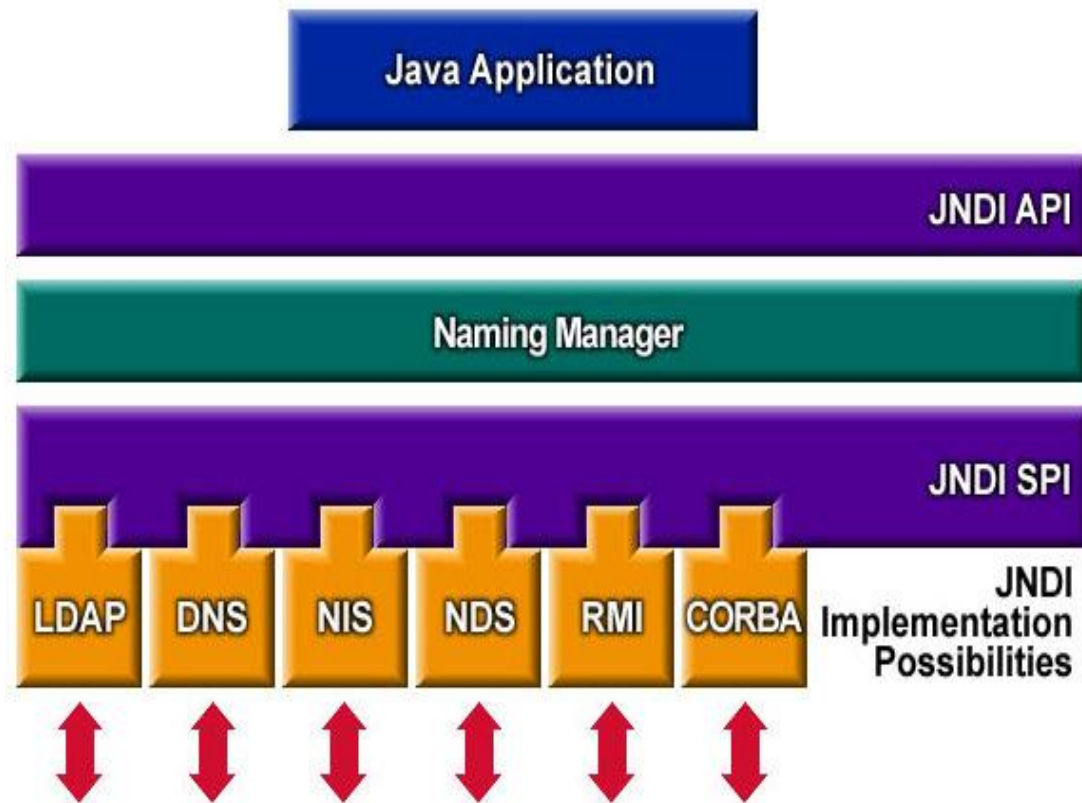
JNDI

- 1 What is JNDI?
- 2 Setup
- 3 Concepts & Classes

What is JNDI?

- Java Naming and Directory Interface API
- Introduced in March, 1997 by Sun Microsystems
- Purpose: to provide a common access to different types of directories

What is JNDI?



JNDI

- The `javax.naming` packages contains mostly Java interfaces.
- Some vendor implements the interface to provide JNDI support for their service.
- To use the service, you need a JNDI Service Provider that implements the interface
- JDK1.4 comes with RMI, DNS, COS, and LDAP Service providers.
- Sun's web site has an additional JNDI Service Provider that works with the local file system

JNDI

- A namespace is a logical space in which names can be defined.
- The same names in different namespaces cause no collisions.
- Namespaces can be nested:
 - file system directories are nested
 - the Internet DNS domains and sub-domains are nested

Packages

`javax.naming`

`javax.naming.directory`

`javax.naming.event`

`javax.naming.ldap`

`javax.naming.spi`

Namespaces are represented by the Context Interface

- Different classes implement this interface differently depending on which naming service they are accessing.
- Has methods to
 - bind and unbind objects to names
 - create and delete sub-contexts
 - lookup and list names
- Since a Context is a Java object it can be registered in another Context with its own name.

The Context Interface

- Start from some “root” context.
- Get the “root” from the InitialContext class
- Examples

LookUp.java

ListCurrentDirectory.java

Class: Context

- **Methods:**

`bind(String name, Object obj);`

`close();`

`list(String name);`

`listBindings(String name);`

`lookup(String name);` // most commonly used

`rebind(String name, Object obj);`

`rename(String oldName, String newName);`

`unbind(String name);`

Class: DirContext

- Extends Context

- methods:

getAttributes(String name);

modifyAttributes(String name, ModificationItem[] mods);

search(String name, Attributes matchAttrs);

Classes: InitialContext & InitialDirContext

- All operations are performed relative to an initial context
- set environment properties
 - Location of server (PROVIDER_URL)
 - How to create a context (INITIAL_CONTEXT_FACTORY)
- instantiation may throw a NamingException

LookUp.java

```
// before running download JNDI provider from Sun  
// add .jar files to classpath
```

```
import javax.naming.Context;  
import javax.naming.InitialContext;  
import javax.naming.NamingException;  
import java.util.Hashtable;  
import java.io.File;
```

```
public class LookUp {  
  
    public static void main(String args[]) throws NamingException {  
  
        try {  
  
            System.out.println("Using a file system (FS) provider");  
  
            // initialize the context with properties for provider  
            // and current directory  
            Hashtable env = new Hashtable();  
            env.put(Context.INITIAL_CONTEXT_FACTORY,  
                    "com.sun.jndi.fscontext.RefFSContextFactory");  
            env.put(Context.PROVIDER_URL,  
                    "file:D:\\McCarthy\\www\\95-702\\examples\\JNDI");  
  
            Context ctx = new InitialContext(env);  
  
            Object obj = ctx.lookup(args[0]);
```

```
if(obj instanceof File) {  
    System.out.println("Found a file object");  
  
    System.out.println(args[0] + " is bound to: " + obj);  
  
    File f = (File) obj;  
  
    System.out.println(f + " is " + f.length() + " bytes long");  
}  
// Close the context when we're done  
ctx.close();  
}  
catch(NamingException e) {  
    System.out.println("Naming exception caught" + e);  
}  
}  
}
```

```
D:\McCarthy\www\95-702\examples\JNDI>java LookUp LookUp.java
```

Using a file system (FS) provider

Found a file object

LookUp.java is bound to: D:\McCarthy\www\95-702\examples\JNDI\LookUp.java

D:\McCarthy\www\95-702\examples\JNDI\LookUp.java is 1255 bytes long

ListCurrentDirectory.java

```
// Use JNDI to list the contents of the current  
// directory
```

```
import javax.naming.Context;  
import javax.naming.InitialContext;  
import javax.naming.NamingException;  
import javax.naming.NamingEnumeration;  
import javax.naming.NameClassPair;  
import java.util.Hashtable;  
import java.io.File;
```

```
public class ListCurrentDirectory {  
  
    public static void main(String args[]) throws NamingException {  
  
        try {  
  
            Hashtable env = new Hashtable();  
            env.put(Context.INITIAL_CONTEXT_FACTORY,  
                    "com.sun.jndi.fscontext.RefFSContextFactory");  
            env.put(Context.PROVIDER_URL,  
                    "file:D:\\McCarthy\\www\\95-702\\examples\\JNDI");  
        }  
    }  
}
```

```
Context ctx = new InitialContext(env);
```

```
NamingEnumeration list = ctx.list(".");
```

```
while (list.hasMore()) {
```

```
    NameClassPair nc = (NameClassPair)list.next();
```

```
    System.out.println(nc);
```

```
}
```

```
ctx.close();
```

```
}
```

```
catch(NamingException e) {
```

```
    System.out.println("Naming exception caught" + e);
```

```
}
```

```
}
```

```
}
```



```
D:\McCarthy\www\95-702\examples\JNDI>java ListCurrentDirectory
ListCurrentDirectory.class: java.io.File
ListCurrentDirectory.java: java.io.File
LookUp.java: java.io.File
SimpleJNDI.java: java.io.File
x: javax.naming.Context
```

```
// Use JNDI to change to a sub directory and list contents
```

```
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.naming.NamingEnumeration;
import javax.naming.NameClassPair;
import java.util.Hashtable;
import java.io.File;

public class ChangeContext {

    public static void main(String args[]) throws NamingException {

        try {
            Hashtable env = new Hashtable();
            env.put(Context.INITIAL_CONTEXT_FACTORY,
                    "com.sun.jndi.fscontext.RefFSContextFactory");
            env.put(Context.PROVIDER_URL,
                    "file:D:\\McCarthy\\www\\95-702\\examples\\JNDI");
```

```
Context ctx = new InitialContext(env);

// a subdirectory called x contains a file f.txt and a subdirectory t
Context sub = (Context)ctx.lookup("x");

NamingEnumeration list = sub.list(".");

while (list.hasMore()) {
    NameClassPair nc = (NameClassPair)list.next();
    System.out.println(nc);
}
ctx.close();
sub.close();

}
catch(NamingException e) {
    System.out.println("Naming exception caught" + e);
}
}
}
```

```
D:\McCarthy\www\95-702\examples\JNDI>java ChangeContext
```

```
f.txt: java.io.File
```

```
t: javax.naming.Context
```

CORBA Transaction Service OTS

X/Open Distributed Transaction Processing

CORBA Transaction Service

CORBA Transaction Service

- CORBA Transaction Service (OTS) provides transaction capabilities to CORBA servers
- The Transactions Service defines interfaces and behaviours that allow multiple, distributed objects to cooperate to provide *ACID* transactions
 - Flat transactions
 - Nested transactions
- OTS is very flexible and provide the developers with a variety of programming models
 - OTS can take care of the most of the work managing a transaction
 - OTS also let you be explicit in how the transaction proceed

ACID

- **ATOMICITY:** A transaction should be done or undone completely and unambiguously. In the event of a failure of any operation, effects of all operations that make up the transaction should be undone, and data should be rolled back to its previous state.
- **CONSISTENCY:** A transaction should preserve all the invariant properties (such as integrity constraints) defined on the data. should preserve all the integrity constraints defined on the data.
- **ISOLATION:** Each transaction should appear to execute independently of other transactions that may be executing concurrently in the same environment. The effect of executing a set of transactions serially should be the same as that of running them concurrently.
- **DURABILITY:** The effects of a completed transaction should always be persistent.

2-Phase Commit Protocol Schema

- Process ordered by the application

- **Messages** exchanged between:

- 1 coordinator
- Several participants

- **2 Phases:**

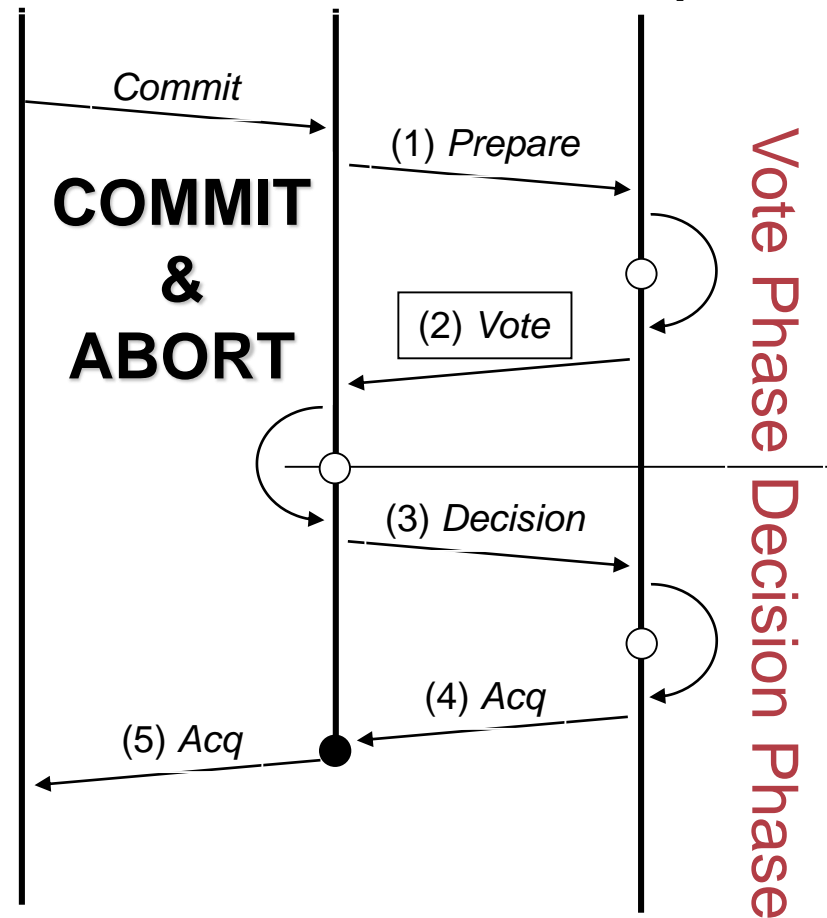
- Vote phase: decide the issue of the transaction
- Decision phase: apply the issue of the transaction

- **Logs** at each phase:

- Journalize the evolution of the protocol

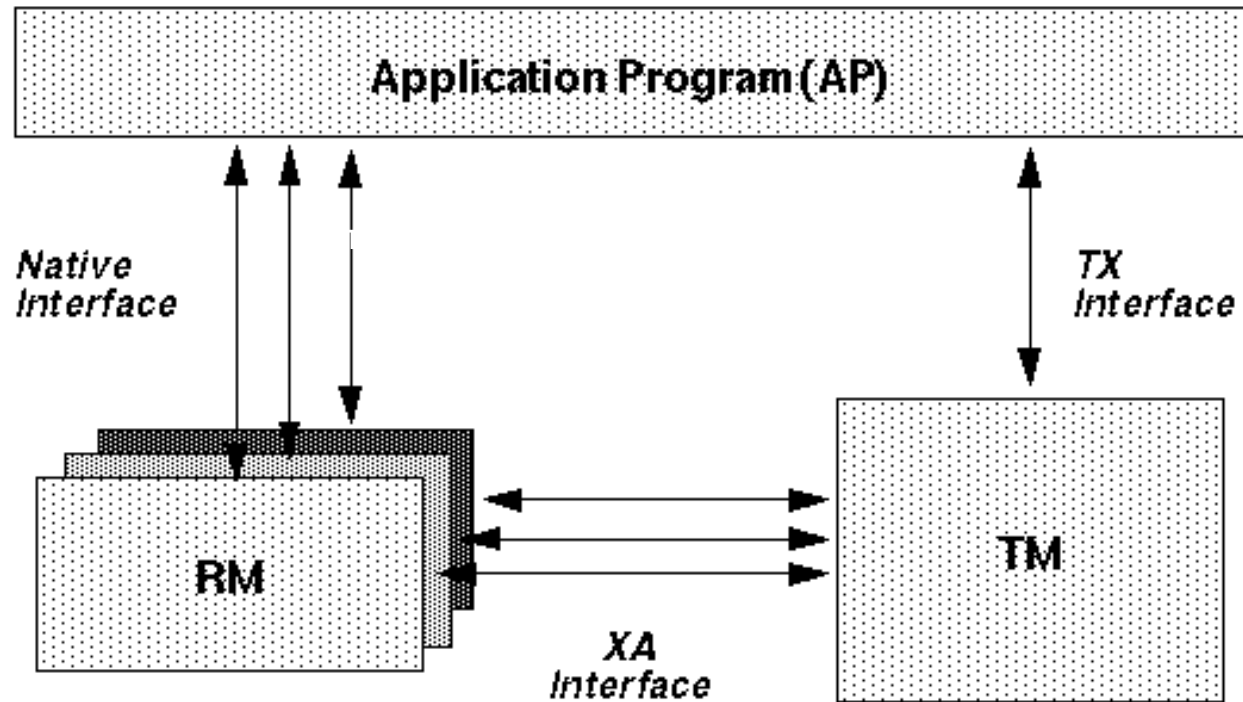
Recovery concern

Application Coordinator Participant

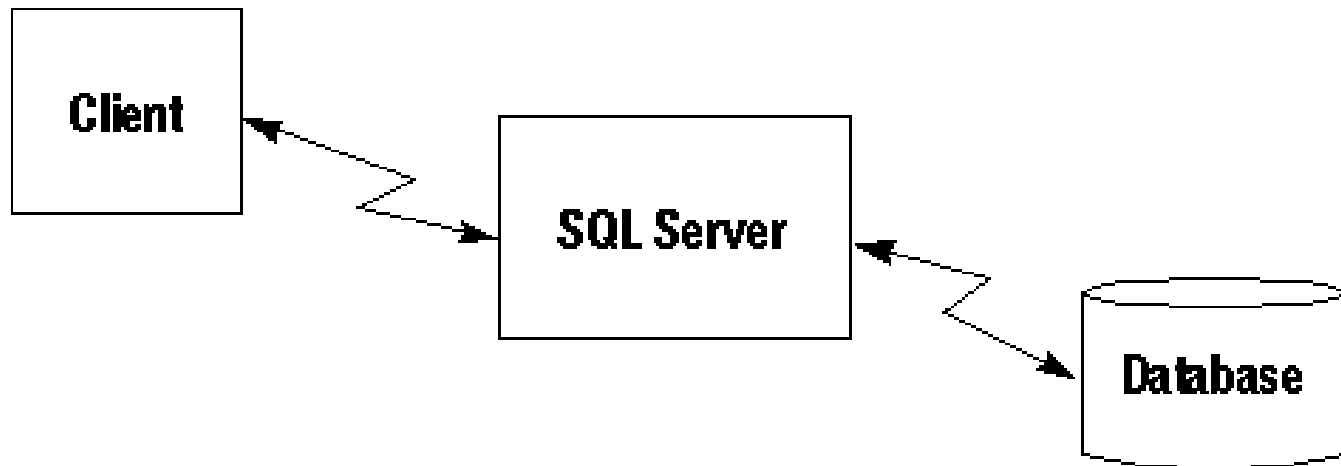


● no force log
○ force log

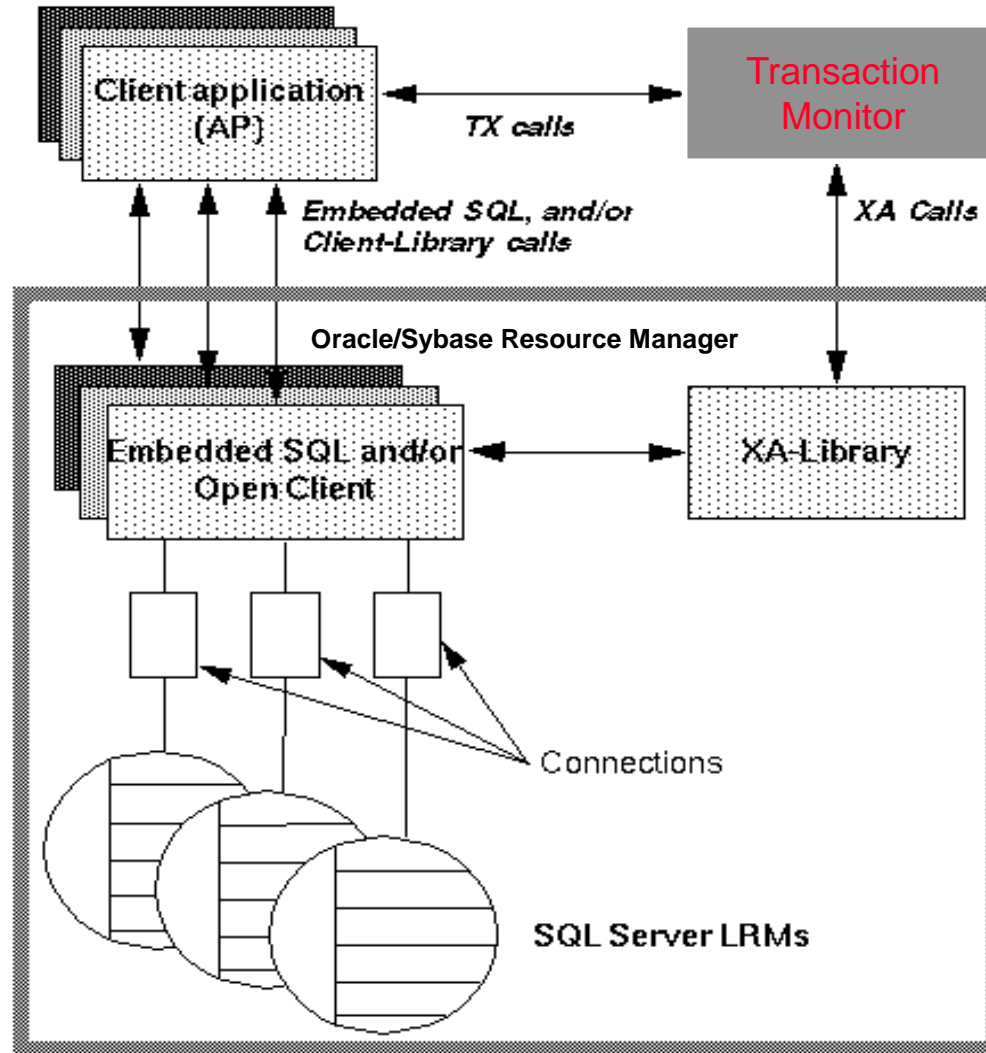
X/Open DTP Model



Transaction Processing Standard Elements



XA DTP Model



TX interface Mapping

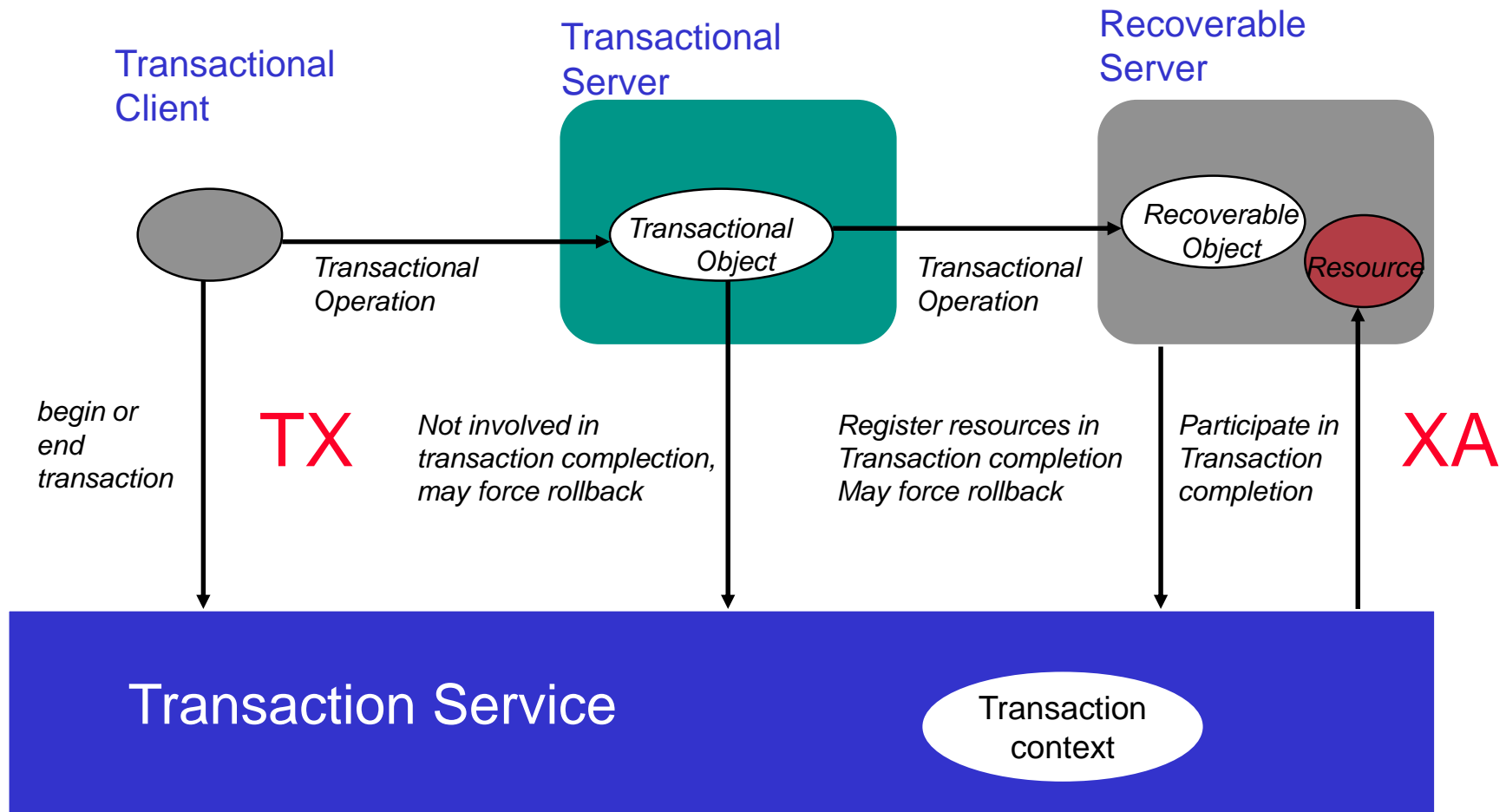
TX interface	Current interface
tx_open()	<i>no equivalent</i>
tx_close()	<i>no equivalent</i>
tx_begin()	Current::begin()
tx_rollback()	Current::rollback() or Current::rollback_only()
tx_commit()	Current::commit()
tx_set_commit_return()	report_heuristics parameter of Current::commit()
tx_set_transaction_control()	<i>no equivalent</i> <i>(chained transactions not supported)</i>
tx_set_transaction_timeout()	Current::set_timeout()
tx_info() - XID	Coordinator::get_txcontext() Current::get_name() ¹
tx_info() - COMMIT_RETURN	<i>no equivalent</i>
tx_info() - TRANSACTION_TIME_OUT	<i>no equivalent</i>
tx_info() - TRANSACTION_STATE	Current::get_status()

XA Interface

XA
A <code>xa_switch_t</code> object
An opened rmid
<code>xa_open_entry(char *, int, long)</code>
<code>xa_close_entry(char *, int, long)</code>
<code>xa_start_entry(XID *, int, long)</code>
<code>xa_end_entry(XID *, int, long)</code>
<code>xa_rollback_entry(XID *, int, long)</code>
<code>xa_prepare(XID *, int, long)</code>
<code>xa_commit(XID *, int, long)</code>
<code>xa_recover(XID *, long, int, long)</code>
<code>xa_forget(XID *, int, long)</code>
<code>xa_complete(int *, int *, int, long)</code>

CORBA Transaction Service

The OTS transaction model



CORBA Transaction Service

The OTS transaction model

- **Transactional Client:** invokes operations on transactional objects and is usually the originator of a transaction
- **Transactional Objects:** participate in transactions and contain or reference data affected by the transaction
 - Inherit from the *TransactionalObject* Interface
 - May contain state that is affected by a transaction but is not recoverable
 - May reference some other recoverable object (its state)

CORBA Transaction Service

- **Recoverable Objects:** is also a transactional object but it directly contains state affected by the transaction
 - Registers an associated *Resource* with the transaction service
 - The Resource stores the persistent state of the recoverable object and participates in the two-phase commit protocol
- **Transactional Servers:** contains one or more transactional objects that are affected by the transaction
 - Interacts with resource managers or other transactional servers

CORBA Transaction Service

The OTS transaction model

- **Recoverable Servers:** contains one or more recoverable objects
 - Has an associated resource that contains persistent data and interacts the protocols of the transaction service
 - The Transaction Service drives the protocol by issuing requests to the resources registered for a transaction
- A client cannot see the difference between a transactional object or a recoverable object – its only concern is that the object is transactional

CORBA Transaction Service

The OTS programming model

- The simple view:
 - The client starts the transaction by contacting the Transaction Service
 - The service creates a transaction context for the new transaction
 - The client then makes a series of requests on transactional or non-transactional objects
 - The client then commits or roll back the changes by instructing the transaction service to commit or rollback
 - The transaction service then coordinate the two-phase commit protocol calling all the registered resources

CORBA Transaction Service

The OTS programming model

- The Transaction Service provides functions for:
 - Control the scope and duration of a transaction
 - Allow multiple objects to be involved in a single atomic transaction
 - Allow objects to associate changes in their internal state with a transaction
 - Coordinate the completion of a transaction using the two-phase commit protocol

CORBA Transaction Service

The OTS programming model

- The client uses the *Current* object to manage transactions (begin, commit, roll back)
- Any participant of the transaction can roll back the transaction – also achieved through the usage of the *Current* object
- If the transactional object is involved in more transactions at a time, it can distinguish between the transactions using a handle to the *Coordinator* (see example coming later)

The OTS programming model - Interfaces

Current

Begin
Commit
Rollback
Rollback_only
Suspend
Resume
Get_status
Get_control
Get_transaction_name

Control

Get_terminator
Get_coordinator

Resource

Prepare
Commit_one_phase
Commit
Rollback
forget

SubtransactionAwareResource

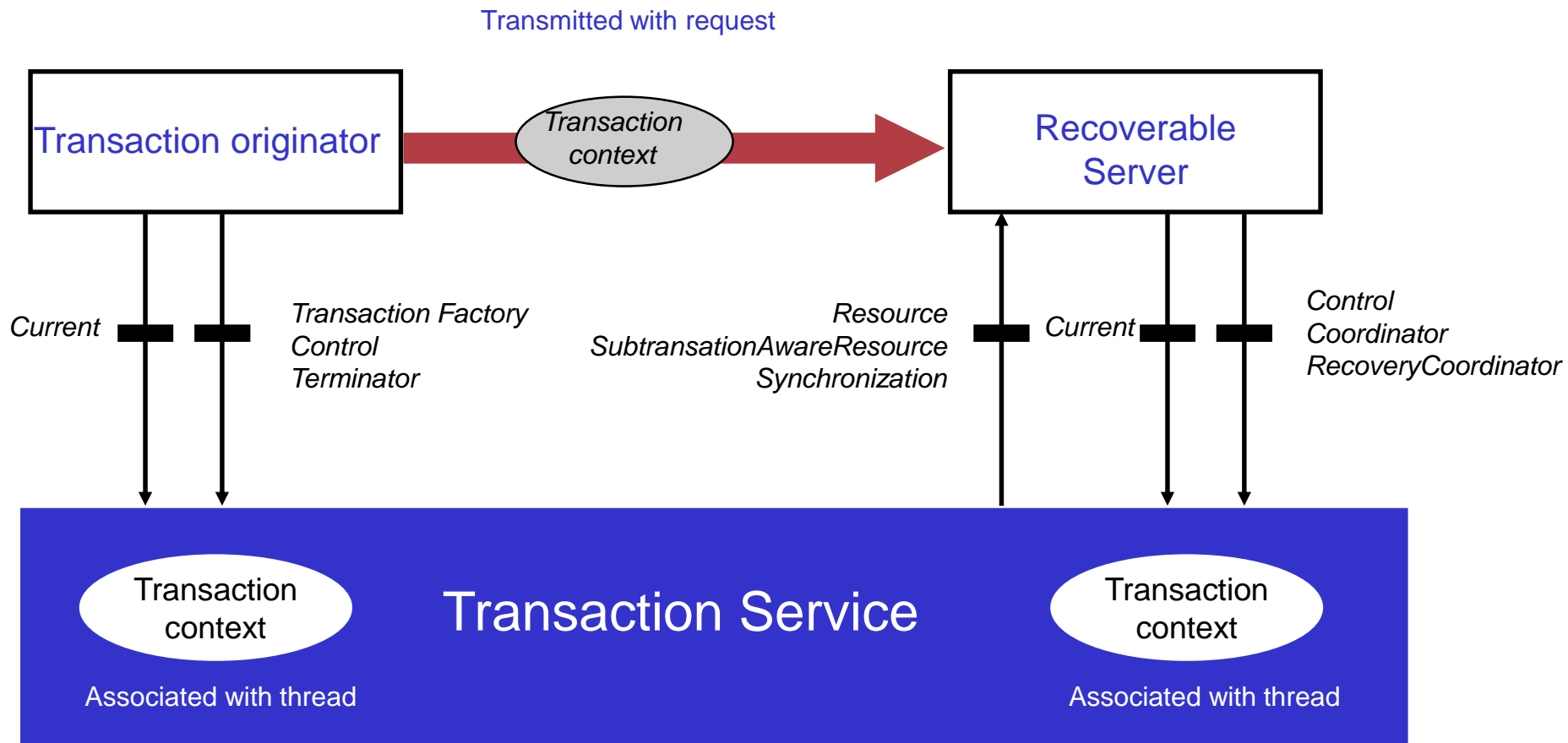
Commit_subtransaction
Rollback_subtransaction

Coordinator

Register_resource
Register_subtranaware
Rollback_only
Get_transaction_name
Create_subtransaction
Get_status
Get_parent_status
Get_top_level_status
Is_same_transaction
Is_related_transaction
Is_ancestor_transaction
Is_decendant_transaction
Is_top_level_transaction
Hash_transaction
Hash_top_level_tran
Get_txcontext

Not all are
listed here

The OTS programming model - Interfaces



CORBA Transaction Service

The OTS programming model

- Objects can support two mechanisms to receive information associated with the transaction (transaction context)
 - **Implicit propagation**: if the object inherits from the *TransactionalObject* interface then a transaction context will be propagated with each method call
 - **Explicit propagation**: the transaction context is propagated by passing the *Control* object as an explicit parameter to methods being invoked

CORBA Transaction Service

The OTS programming model

- Participants can choose two ways for transaction context management (transaction context creation, initialization and control)
 - **Indirect context management**: rely on the Transaction Service to perform all context related responsibilities
 - **Direct context management**: elect to manage the context directly by using the standard OTS interfaces (*Control*, *TransactionFactory*, *Terminator*)

Simplyfies
code

May be more
convenient
for explicit
propagation

CORBA Transaction Service

The OTS programming model

- Example of a transactional server:
 - start Naming Server (startnamingservice)
 - start Transaction Service (Just a POA server) (starttransactionservice)
 - start bank server (startwoserver)
 - start one bank client (startwoclient)
 - show effect of roolback by making a transfer which is not allowed
 - no atomicity

CORBA Transaction Service

The OTS programming model

- When making recoverable resources you have two choices:
 - You can use a database or another form of persistent storage for which there is a resource manager compliant with XA standard or native OTS support
 - You can implement your own recoverable resource
 - Wrapper around some non-XA compliant persistent storage
 - Completely your own code

CORBA Transaction Service

The OTS programming model

- When making recoverable objects, The Transaction Service controls the resource objects
- Each transactional object has to register the resource for a transactional object when the transaction is started
 - More clients concurrently working on the same transactional object are given a copy (the resource object) which is controlled by the Transaction Service
 - Upon commit the transactional object is given the values from the resource object (up to the developer to make this happen)

CORBA Transaction Service

The OTS programming model

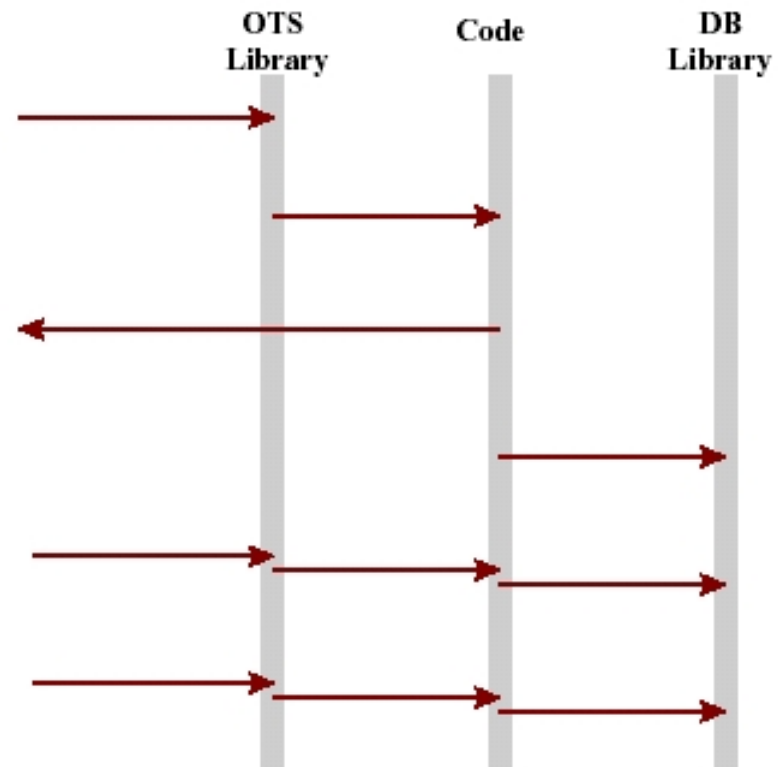
- The resource registration process can either manual or automatic
- In the manual case you use the **Coordinator** pseudo object implementing the *Coordinator* interface
- Some vendors however cooperate with database vendors to make databases have native support for the **Resource** interface

CORBA Transaction Service

The OTS programming model – Manual resource registration

In the case of explicit transaction Context propagation

- Invocation Arrives, intercepted by OTS
- Invocation propagated to user code
- Code registers its own resource
- SQL update to database
- Prepare call arrives, code calls DB
- Commit call arrives, code calls DB

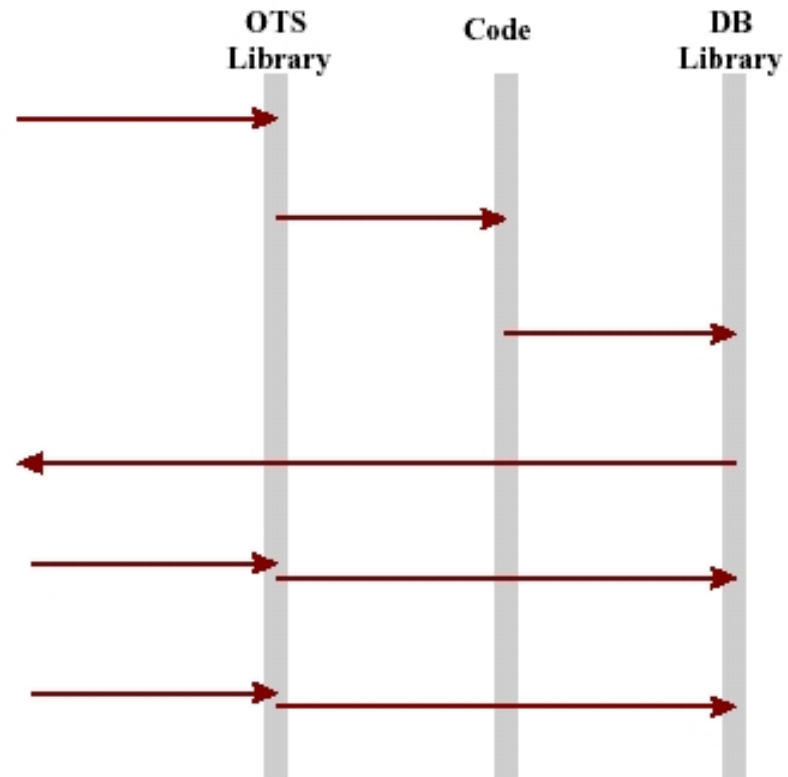


CORBA Transaction Service

The OTS programming model – Automatic resource registration

In the case of implicit transaction
Context propagation

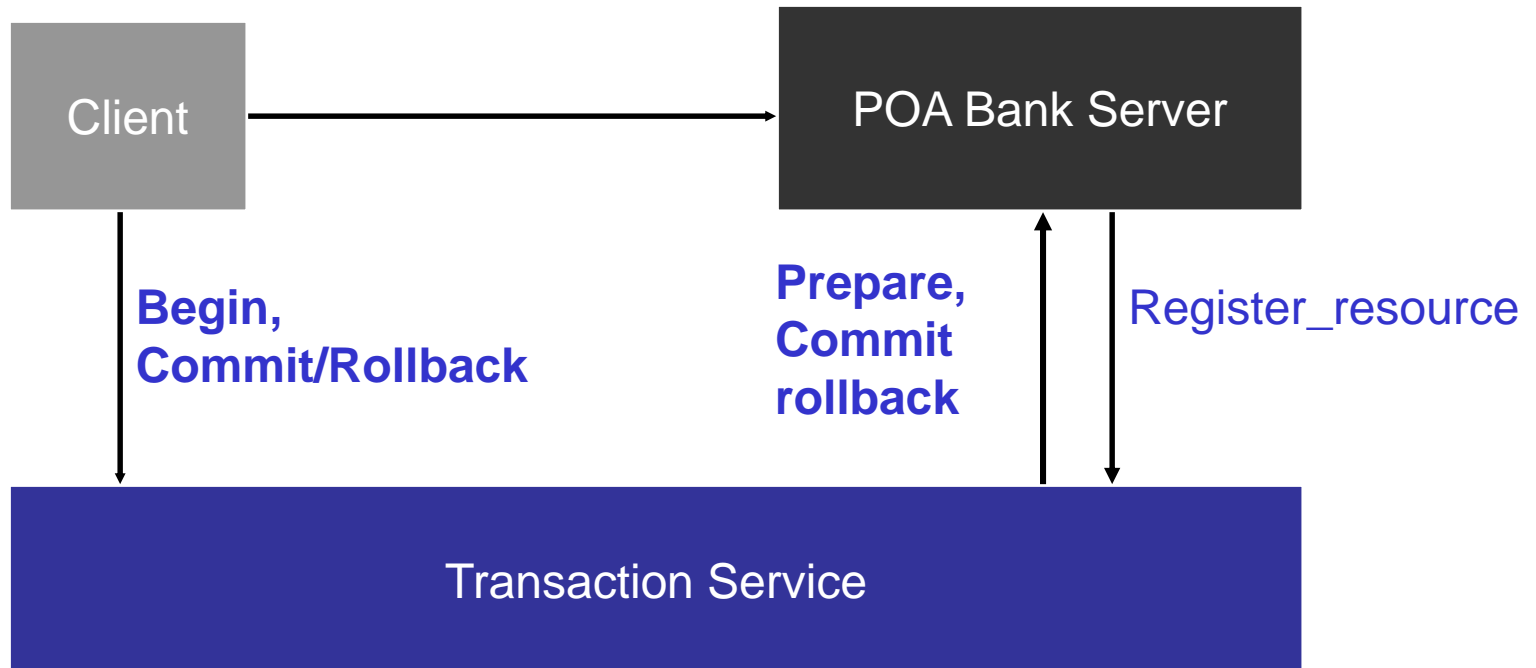
- Invocation Arrives, intercepted by OTS
- Invocation propagated to user code
- SQL update to database
- Database registers its native *Resource*
- Prepare call arrives
- Commit call arrives



The OTS programming model - Example

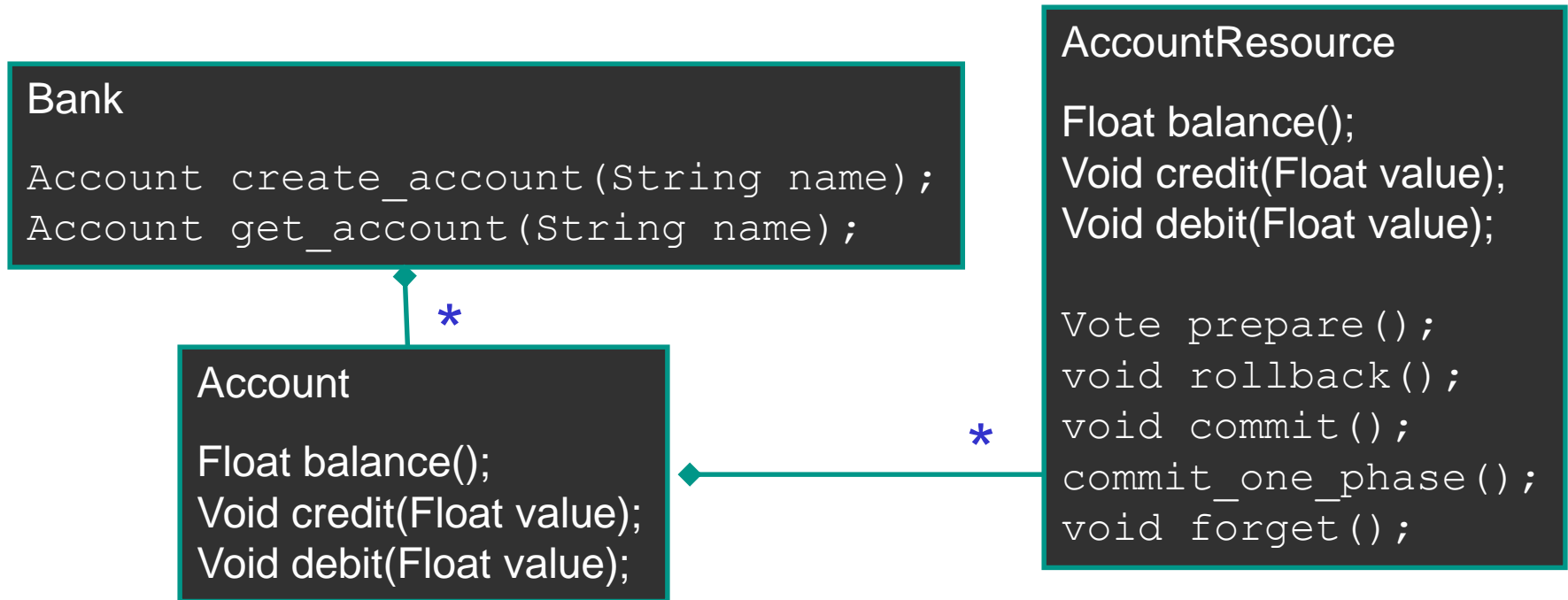
- Example of a (not persistent) recoverable server:
 - start Naming Server (startnamingservice)
 - start Transaction Service (Just a POA server) (starttransactionservice)
 - start new bankserver (startwserver)
 - start one bank client (startwclient)
 - show effect of rollback by making a transfer which is not allowed
 - atomicity achieved
 - Show the effect of a server crash in the middle of a transaction
 - Server is NOT recoverable

The OTS programming model - Example



The OTS programming model - Example

Registered in the Naming Service – factory for accounts



For each Account instance (identified by name) there will be a number of AccountResource instances aggregated (one for each transaction), all which are registered as resources in the Transaction Service server

The OTS programming model - Example

```
public class BankImpl extends BankPOA
{
    private java.util.Hashtable _accounts;
    ...
    public Account create_account( String name ) {
        AccountImpl acc = new AccountImpl( _orb, name );
        _accounts.put( name, acc );
        return acc._this( _orb );
    }

    public Account get_account( String name )
        throws NotExistingAccount {
        AccountImpl acc = ( AccountImpl ) _accounts.get( name );
        if ( acc == null )
            throw new NotExistingAccount();
        return acc._this( _orb );
    }
}
```

The OTS programming model - Example

```
public class AccountImpl extends AccountPOA {  
    ...  
    public float balance() {  
        return get_resource().balance();  
    }  
    public AccountResource get_resource() {  
        try {  
            Object obj = _orb.resolve_initial_references( "TransactionCurrent" );  
            Current current = org.omg.CosTransactions.CurrentHelper.narrow( obj );  
            Coordinator coordinator = current.get_control().get_coordinator();  
            AccountResourceImpl resource = null;  
            for ( int i = 0; i < _resources.size(); i++ ) {  
                resource = ( AccountResourceImpl ) _resources.elementAt( i );  
                if ( coordinator.is_same_transaction( resource.coordinator() ) )  
                    return resource._this( _orb );  
            }  
            resource = new AccountResourceImpl( _poa(), coordinator, this, _name );  
            AccountResource res = resource._this( _orb );  
            coordinator.register_resource( res );  
            _resources.addElement( resource );  
            return res;  
        } catch (.....) ...  
    }  
}
```

Clients share the same account instances but requests are routed to an AccountResource which are created for each transaction


The Transaction Service is just a POA server which registers remote callback objects

The OTS programming model - Example


```
public class AccountResourceImpl extends AccountResourcePOA {
    ...
    private AccountImpl _account;

    public AccountResourceImpl( org.omg.PortableServer.POA poa,
                               org.omg.CosTransactions.Coordinator coordinator,
                               AccountImpl account, String name ){
        _account = account;
        .....
    }
    .....
    public void credit( float value ) {
        _current_balance += value;
    }

    public void commit() throws org.omg.CosTransactions.NotPrepared,
                               org.omg.CosTransactions.HeuristicRollback,
                               org.omg.CosTransactions.HeuristicMixed,
                               org.omg.CosTransactions.HeuristicHazard {
        _account._balance = _current_balance;
        removeItself();
    }
}
```



The transactional Object
Can be referenced

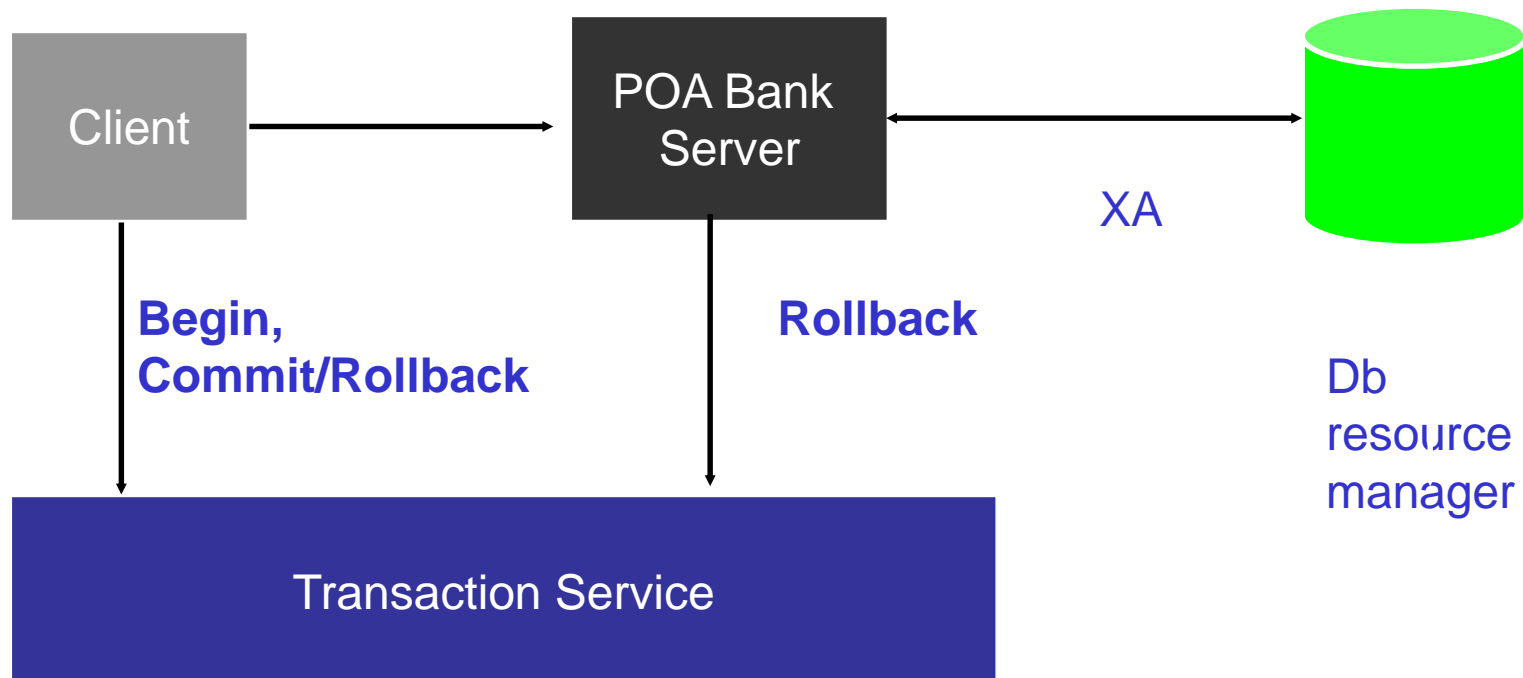


By commit the trans-
actional object is up-
dated

The OTS programming model - Example

- Example of using a XA compliant database (Oracle) to make a recoverable object server:
 - start Naming Server (startnamingservice)
 - start Transaction Service (Just a POA server) (starttransactionservice)
 - start new bankserver (startwxaserver)
 - start one bank client (startwxacient)
 - show effect of rollback by making a transfer which is not allowed
 - atomicity achieved
 - Show the effect of a server crash in the middle of a transaction
 - Server is recoverable

The OTS programming model – Example II

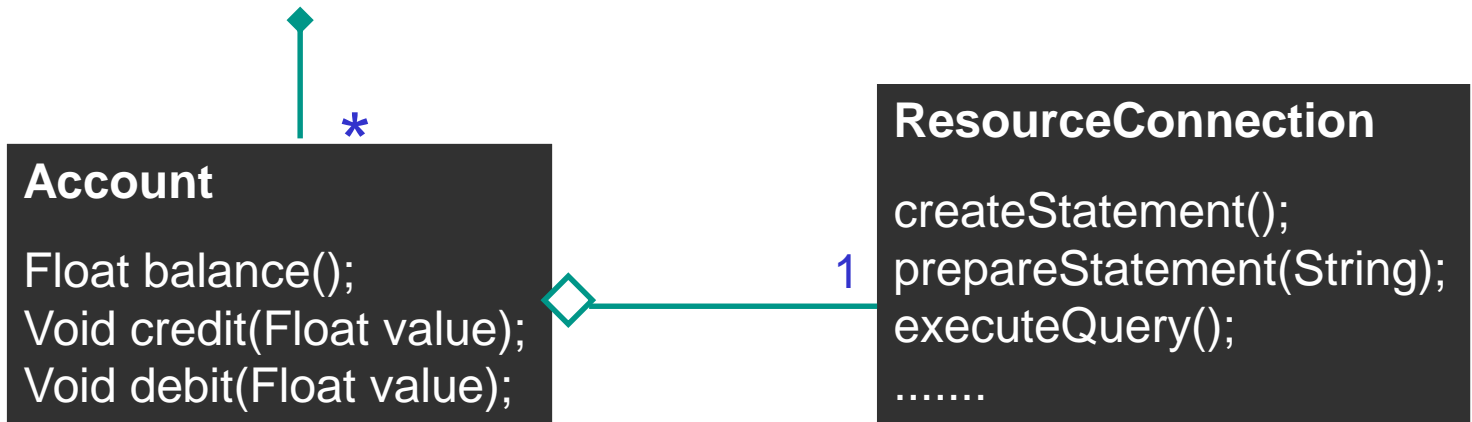


The OTS programming model – Example II

Registered in the Naming Service – factory for accounts

Bank

```
Account create_account(String name);  
Account get_account(String name);
```



For each Account instance (identified by name) there will be a database connection aggregated acting as the resource
No resources are registered in the Transaction Service server

The OTS programming model – Example II

```
public class BankImpl extends BankPOA {
    private java.sql.Connection _db;
    private java.util.Hashtable _accounts;

    ...
    public Account create_account( String name )
        throws AlreadyExistingAccount {
        connectToDatabase();
        AccountImpl acc = new AccountImpl( _orb, name, _db );
        _accounts.put( name, acc );
        try {
            java.sql.PreparedStatement pstmt =
                _db.prepareStatement("INSERT INTO Accounts VALUES (?, ?)");
            pstmt.setString(1, name);
            pstmt.setFloat(2, 0);
            pstmt.executeUpdate();
            pstmt.close();
        } catch ( .... ) { .. }
        return acc._this( _orb );
    }
}
```

Connect to persistent
resource

Access
Persistent
resource

Never do this:
-factor out persistence code
-Call stored procedure instead
of insert (or something else)

No commit as the Transaction Service
is responsible for this

The OTS programming model – Example II

```
public Account get_account( String name )throws NotExistingAccount {
    AccountImpl acc = ( AccountImpl ) _accounts.get( name );
    if ( acc == null ) {
        connectToDatabase();
        java.sql.ResultSet res = null;
        try {
            PreparedStatement pstmt =
                _db.prepareStatement("SELECT balance from Accounts where name=?");
            pstmt.setString( 1, name );
            res = pstmt.executeQuery();
            if (res.next()) {
                res.getFloat(1);
                acc = new AccountImpl( _orb, name, _db );
                _accounts.put( name, acc );
            }
            pstmt.close();
        } catch ( .. ) {..}
    }
    if ( acc == null ) throw new NotExistingAccount();
    return acc._this( _orb );
}
```

The OTS programming model – Example II

```
public void connectToDatabase() {  
    try {  
        Object obj = _orb.resolve_initial_references  
                        ("TransactionSessionManager");  
        SessionManager session =  
            org.openorb.ots.SessionManagerHelper.narrow(obj);  
        _db = session.getConnection("test","test","InstantDB");  
        _db.setTransactionIsolation(TRANSACTION_READ_COMMITTED);  
    } catch ( ... ) {...}  
  
    if ( !tableExist() )  
        createTable();  
}  
....  
}
```

Calls the database through the
sessionManager via XA

Oracle JDBC 2.0 driver supports
The XA interface

The OTS programming model – Example II

```
public class AccountImpl extends AccountPOA {
    ...
    public float balance()    {
        java.sql.ResultSet res = null;
        java.sql.PreparedStatement pstmt = null;
        try {
            pstmt=_db.prepareStatement("SELECT balance from Accounts where name=?");
            pstmt.setString( 1, _name );
            res = pstmt.executeQuery();
            res.next();
            return res.getFloat( 1 );
        }catch (...) {}
        finally {... //close pstmt}
        return -1;
    }
    public void credit( float value ) {
        float _balance = balance();
        _balance += value;
        updateBalance( _balance );
    }
}
```

The OTS programming model – Example II

AccountImpl continued

```
public void debit( float value ){
    float _balance = balance();
    _balance -= value;
    if ( _balance < 0 ) rollbackOnly();
    updateBalance( _balance );
}

public void updateBalance( float _balance ) {
    try {
        PreparedStatement pstmt =
            _db.prepareStatement("UPDATE Accounts
                                SET balance=? WHERE name=?");

        pstmt.setFloat( 1, _balance );
        pstmt.setString( 2, _name );
        pstmt.executeUpdate();
        pstmt.close();
    } catch ( ... ) {..}
}
```

No commit as the
Transaction Service is
Responsible for this

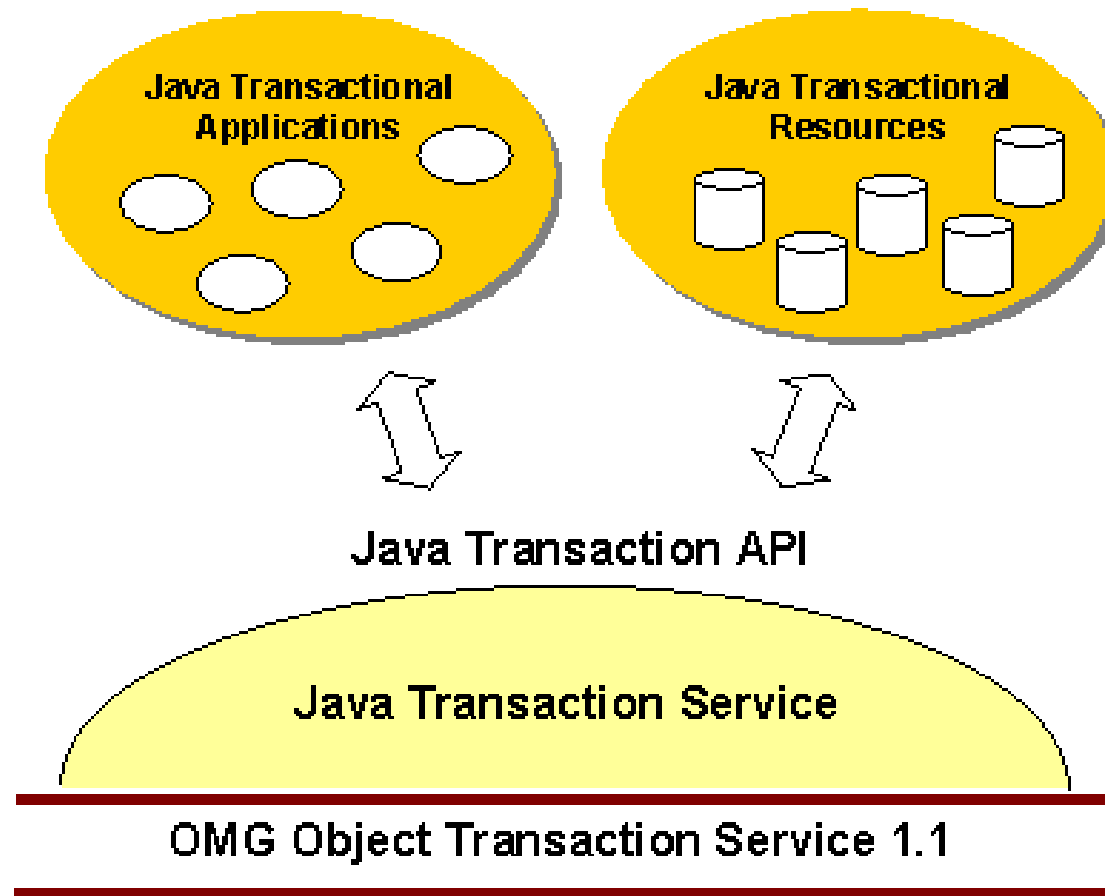
The OTS programming model – Example II

AccountImpl continued

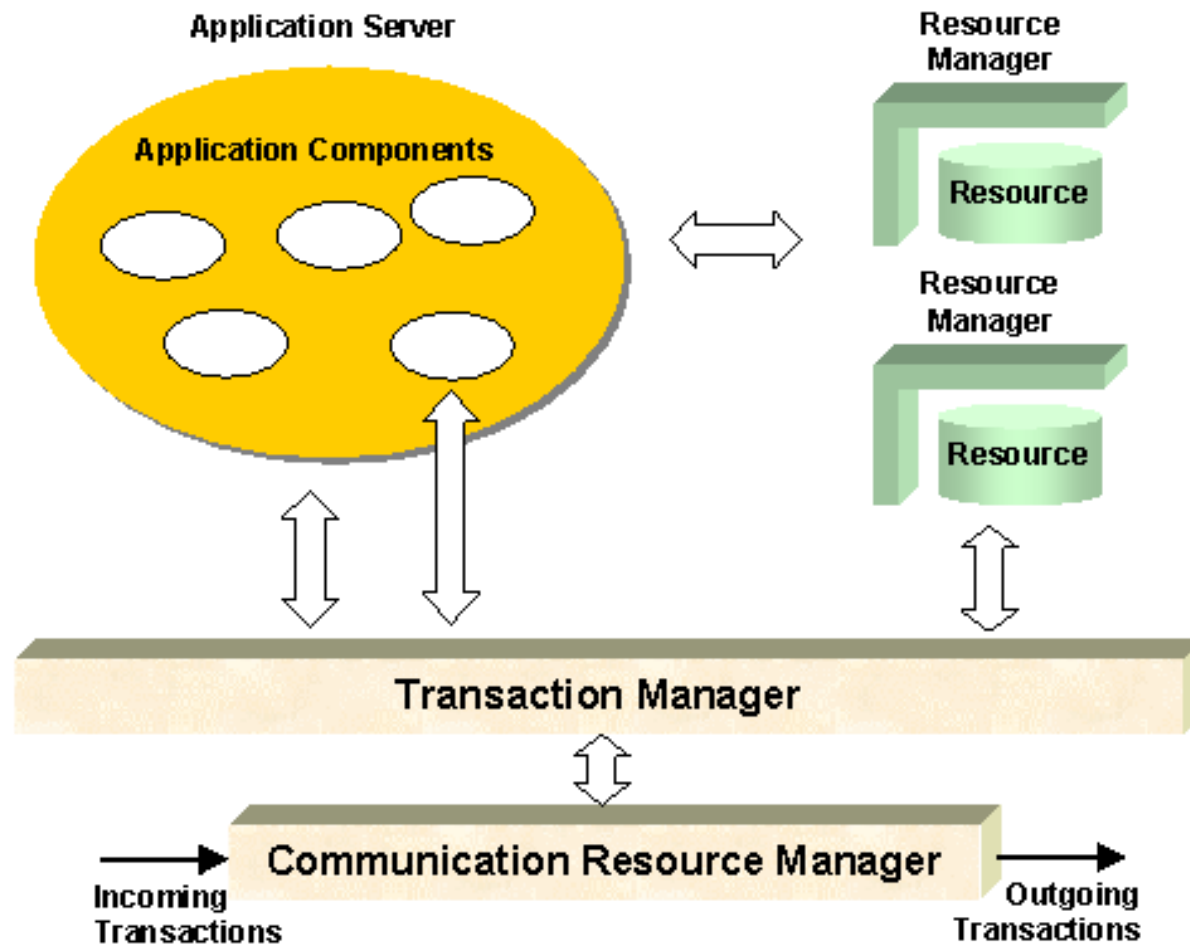
```
public void rollbackOnly() {  
    try {  
        Object obj=_orb.resolve_initial_references("TransactionCurrent");  
        Current current=org.omg.CosTransactions.CurrentHelper.narrow(obj);  
        current.rollback_only();  
    } catch (.... ) { .. }  
}
```

The method `rollback()` are in some implementations of Transaction Service only for the transaction originator client therefore `rollback_only` is used here as it is available to all transaction participants

Java Transaction Service JTS








JTS Architecture



Java Transaction API

Java Transaction Service

-  `javax.transaction.Status`
-  `javax.transaction.Transaction`
-  `javax.transaction.TransactionManager`
-  `javax.transaction.UserTransaction`
-  `javax.transaction.xa.Xid`

Resource Manager

-  `javax.transaction.xa.XAResource`

Resource Manager

-  `javax.transaction.Synchronization`

OTS vs. JTA

Transaction Service	JTA
Current	<code>TransactionManager</code> , <code>UserTransaction</code>
Synchronization	<code>Synchronization</code>
Control/Coordinator/Terminator	<code>Transaction</code>
Current::begin	<code>TransactionManager.begin()</code> , <code>UserTransaction.begin()</code>
Current::commit	<code>TransactionManager.commit()</code> , <code>UserTransaction.commit()</code>

CORBA Transaction Service

Data Isolation

- For pure OTS applications using the *Resource* interface data isolation is best achieved by using the CORBA Concurrency Service

Later we will see how to use CCS to develop data isolation

- If you are using a database as your recoverable server, the database will/can take care of the data isolation, but you can also combine it with CCS
 - The OTS Transaction Service will act as Transaction manager and coordinate the two-phase commit protocol with the resource manager for the database

CORBA Transaction Service

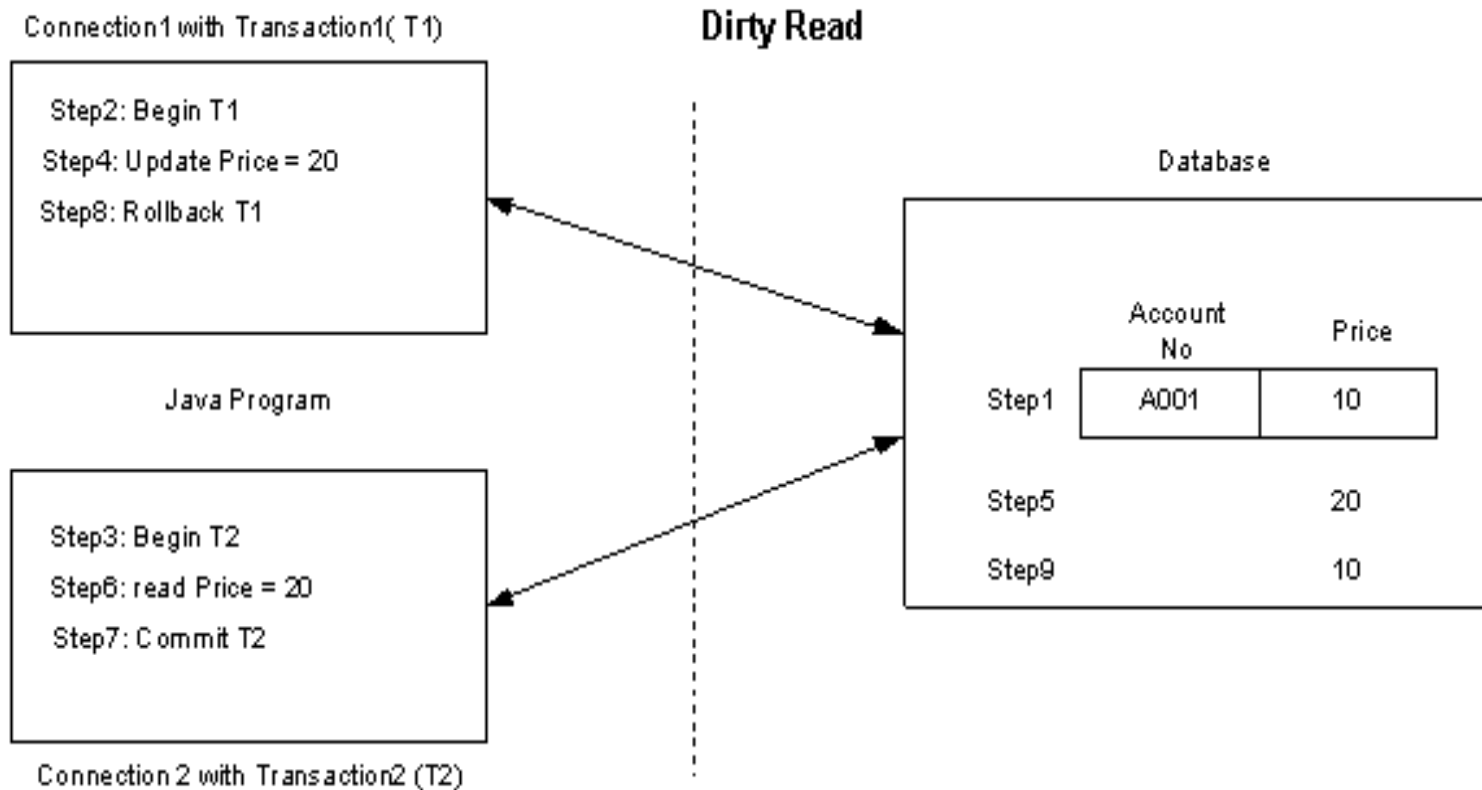
Data Isolation

- Two typical ways of making your XA Recoverable server resistant for concurrent users
 - Adapt an Optimistic locking scheme or
 - Adapt a Pessimistic locking scheme
 - Each client gets its own Account instance and data isolation is delegated to the Database – no sharing of transactional objects
- Or serialize access to Account object with CCS

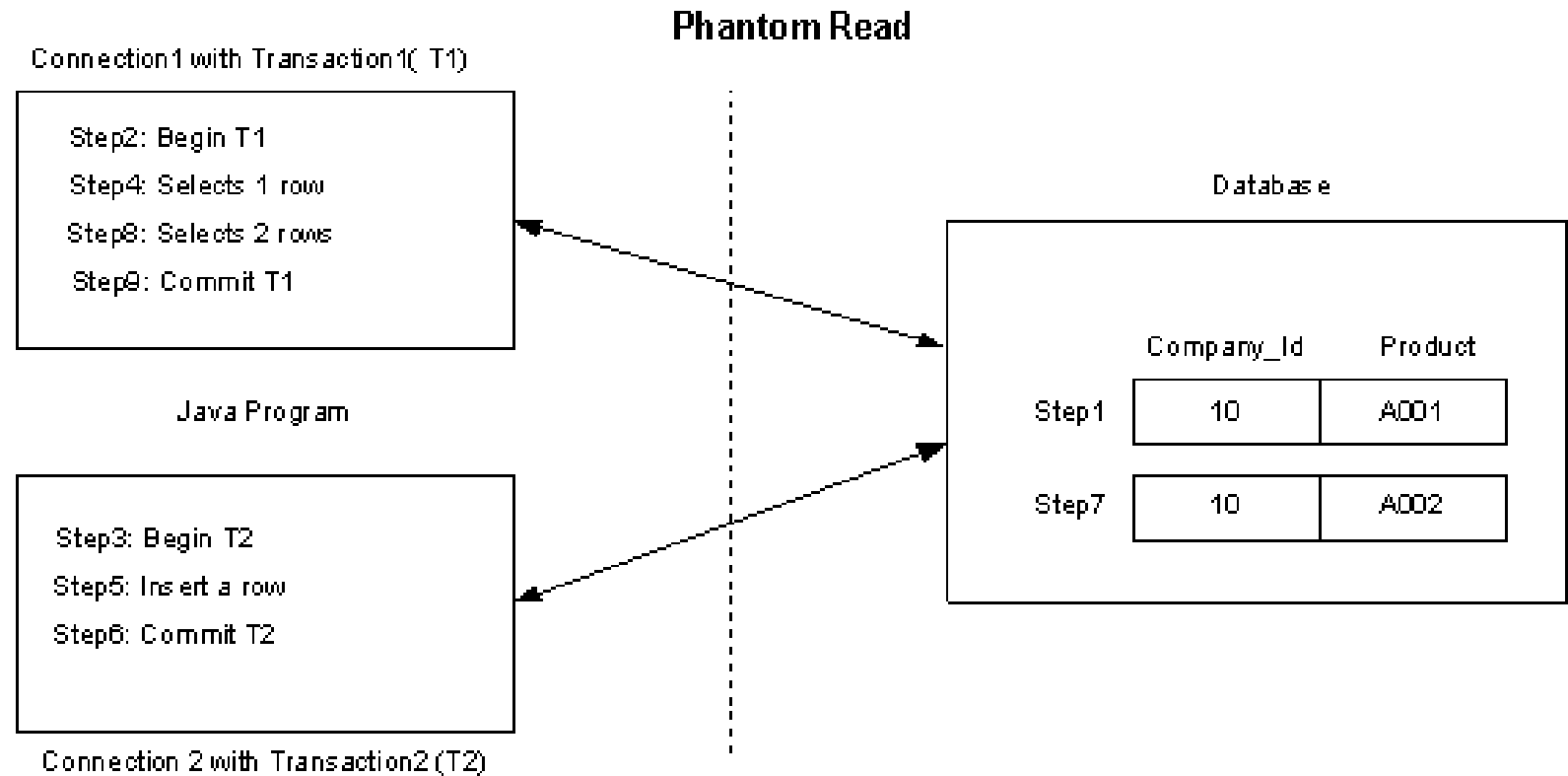
Isolation Levels

Transaction Level	Permitted Phenomena			Performance Impact
	Dirty Reads	Non Repeatable Reads	Phantom Reads	
TRANSACTION_READ_UNCOMMITTED	YES	YES	YES	Fastest
TRANSACTION_READ_COMMITTED	NO	YES	YES	Fast
TRANSACTION_REPEATABLE_READ	NO	NO	YES	Medium
TRANSACTION_SERIALIZABLE	NO	NO	NO	Slow

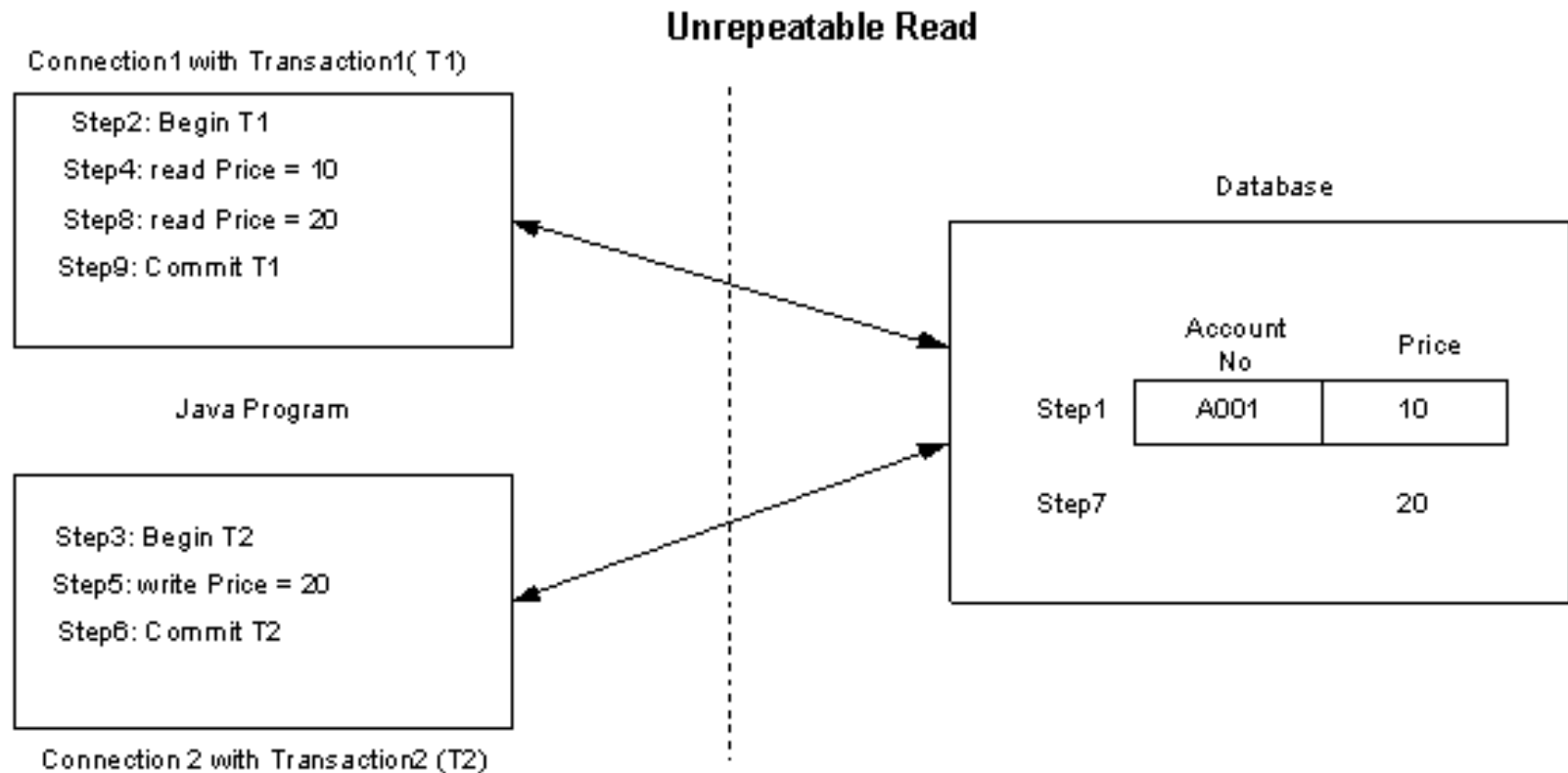
Dirty Read



Phantom Read



Unrepeatable Read



CORBA Transaction Service

Data Isolation - Isolation for Oracle Database System

- The Oracle Database provides three isolation levels
 - Read Committed
 - A transaction reads only committed data
 - Each query sees only data that is committed before execution of the query began
 - Serializable
 - A transaction sees only data that was committed before the transaction began plus changes made by the transaction itself
 - Access to data is serialized

CORBA Transaction Service

Data Isolation - Isolation for Oracle Database System

- Read-Only
 - The transaction only sees data as there were when the transaction started
 - Updates not allowed

CORBA Concurrency Service

- "The purpose of the Concurrency Control Service is to mediate concurrent access to an object such that the consistency of the object is not compromised when accessed concurrently" (CCS spec)
- Can be used in two ways
 - Transactional: on behalf of a transaction -> transaction service responsible for releasing locks as a part of the transaction completion
 - Non-transactional: outside transactions on behalf of the current thread -> user of CCS is responsible for releasing locks

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data
- Two types of locks:
 - *Shared* (S) locks (also called *read locks*)
 - Obtained if we want to only read an item
 - *Exclusive* (X) locks (also called *write locks*)
 - Obtained for updating a data item

Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
 - It decides whether to *grant* a lock request
- T1 asks for a lock on data item A, and T2 currently has a lock on it ?
 - Depends

<u>T2 lock type</u>	<u>T1 lock type</u>	<u>Should allow ?</u>
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

- If *compatible*, grant the lock, otherwise T1 waits in a *queue*.

CORBA Concurrency Service

- Lock model of CORBA Concurrency Control Service

*Lock conflict **

Granted Mode	Requested Mode				
	<i>IR</i>	<i>R</i>	<i>U</i>	<i>IW</i>	<i>W</i>
Intention Read (IR)					*
Read (R)				*	*
Upgrade (U)			*	*	*
Intention Write (IW)		*	*		*
Write (W)	*	*	*	*	*

You place locks on resources like fx Objects (FIFO)

An **Upgrade read** lock is to avoid deadlocks

Intention write lock is to do variable Lock granularity in hierachical Relationships – set on ancestor

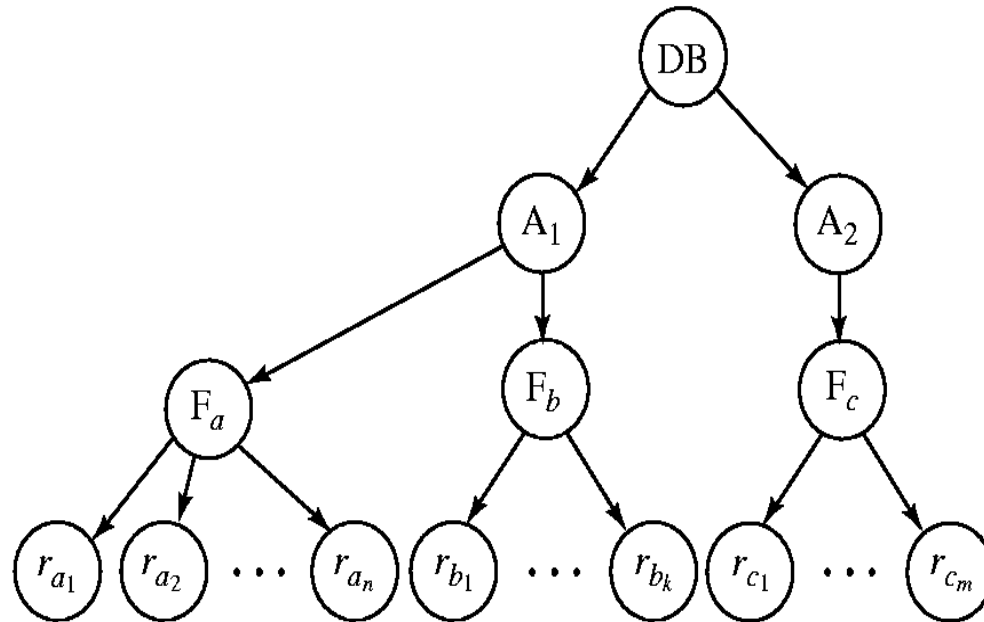
Intention read lock - do.

CORBA Concurrency Service

- The granularity of locks determine the concurrency in the application
 - Coarse-grained locks -> less concurrency and low overhead
 - Fine-grained locks -> more concurrency but high overhead

Database Table Record

Granularity Hierarchy



The highest level in the example hierarchy is the entire database.

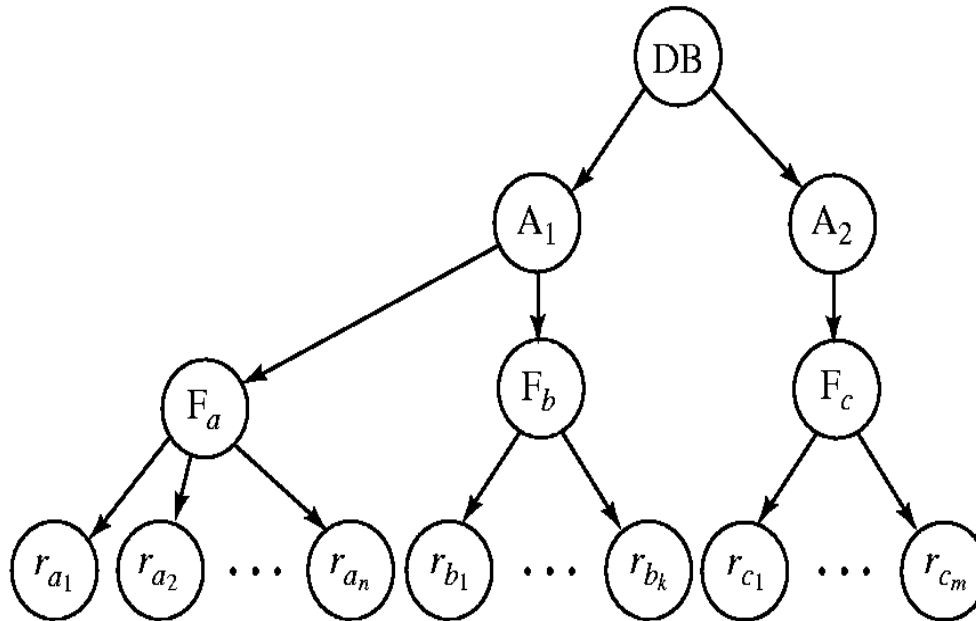
The levels below are of type *area*, *file* or *relation* and *record* in that order.

Can look at any level in the hierarchy

Intention Lock

- New lock mode, called *intentional* locks
 - Declare an intention to lock parts of the subtree below a node
 - **IS: *intention shared***
 - The lower levels below may be locked in the shared mode
 - **IX: *intention exclusive***
 - **SIX: *shared and intention-exclusive***
 - The entire subtree is locked in the shared mode, but I might also want to get exclusive locks on the nodes below
- Protocol:
 - If you want to acquire a lock on a data item, all the ancestors must be locked as well, at least in the intentional mode
 - So you always start at the top *root* node

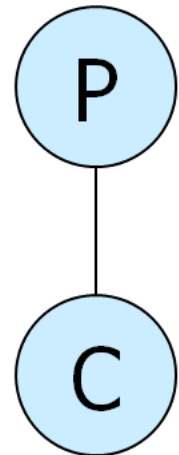
Granularity Hierarchy



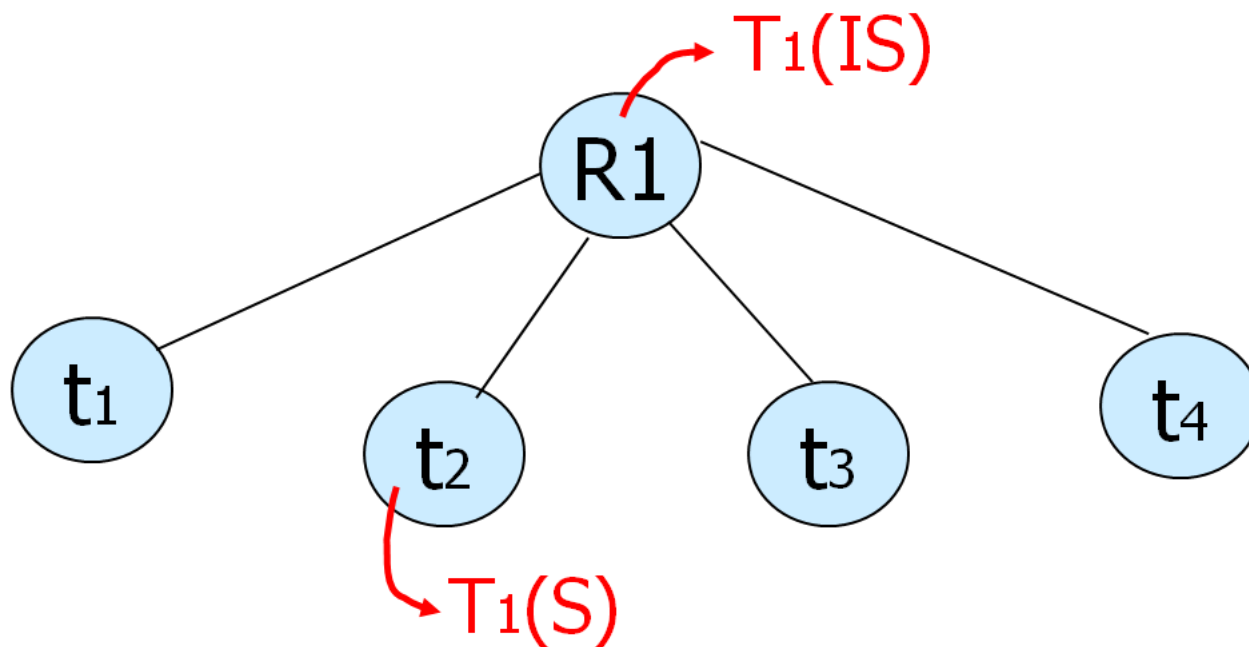
- (1) Want to lock F_a in shared mode, DB and A_1 must be locked in at least IS mode (but IX, SIX, S, X are okay too)
- (2) Want to lock rc_1 in exclusive mode, DB , A_2 , F_c must be locked in at least IX mode (SIX, X are okay too)

Granularity Hierarchy

Parent locked in	Child can be locked in
IS	IS, S
IX	IS, S, IX, X, SIX
S	[S, IS] not necessary
SIX	X, IX, [SIX]
X	none

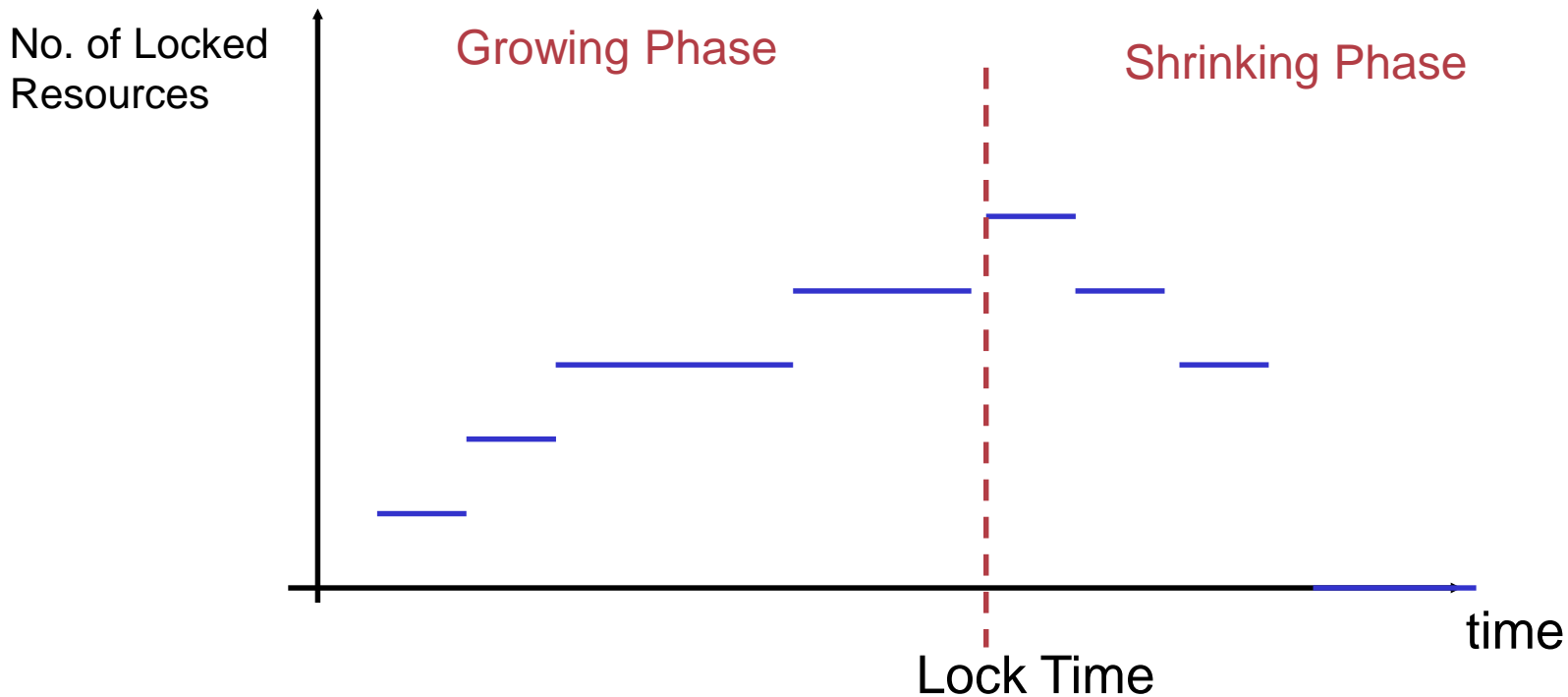


Example



CORBA Concurrency Service

Two Phase Locking Protocol - 2PL



2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
 - Transaction may obtain locks
 - But may not release them
- Phase 2: Shrinking phase
 - Transaction may only release locks
- Can be shown that this achieves *conflict-serializability*
 - lock-point: the time at which a transaction acquired last lock
 - if lock-point(T1) < lock-point(T2), there can't be an edge from T2 to T1 in the *precedence graph*

T1

lock-X(B)
read(B)
 $B \leftarrow B - 50$
write(B)
unlock(B)

lock-X(A)
read(A)
 $A \leftarrow A + 50$
write(A)
unlock(A)

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort
 - Read locks are not important
- Rigorous 2PL: Release both *exclusive and read* locks only at the very end
 - The serializability order === the commit order
 - More intuitive behavior for the users
 - No difference for the system