

Chapter 4 Remote Procedure Calls and Distributed Transactions



Middleware for Heterogenous and Distributed Information Systems - WS06/07

Outline

- Remote Procedure Call
 - concepts
 - IDL, principles, binding
 - variations
 - remote method invocation
 - example: Java RMI
 - stored procedures
- Distributed Transaction Processing
 - transactional RPC
 - X/Open DTP
- Summary



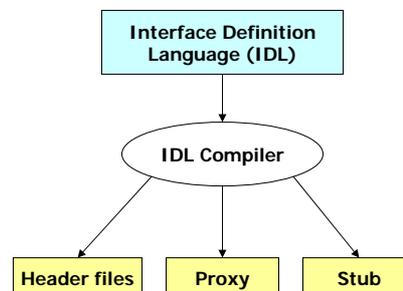
Communication and Distributed Processing

- Distributed (Information) System
 - consists of (possibly autonomous) subsystems
 - jointly working in a coordinated manner
- How do subsystems communicate?
 - **Remote Procedure Calls (RPC)**
 - transparently invoke procedures located on other machines
 - Peer-To-Peer-Messaging
 - Message Queuing
- Transactional Support (ACID properties) for distributed processing
 - Server/system components are Resource Managers
 - (Transactional) Remote Procedure Calls (TRPC)
 - Distributed Transaction Processing



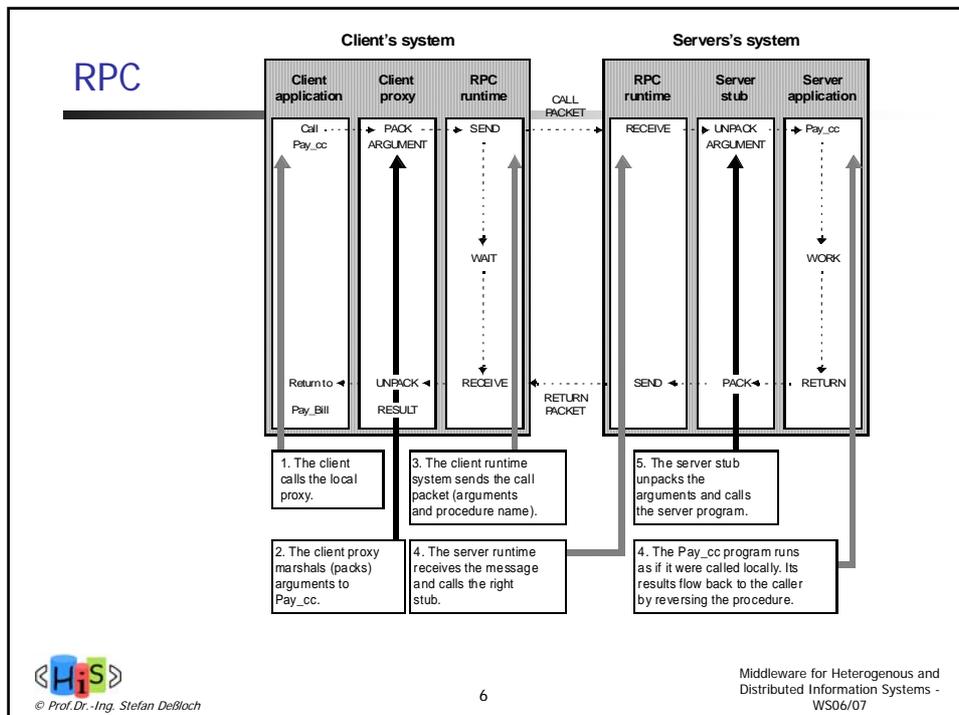
Remote Procedure Call (RPC)

- Goal: Simple programming model for distributed applications
 - based on procedure as an invocation mechanism for distributed components
- Core mechanism in almost every form of middleware
- Distributed programs can interact (transparently) in heterogeneous environments
 - network protocols
 - programming languages
 - operating systems
 - hardware platforms
- Important concepts
 - Interface Definition Language (IDL)
 - Proxy (Client Stub)
 - Stub (Server Stub)



How RPC Works

- Define an interface for the remote procedure using an IDL
 - abstract representation of procedure
 - input and output parameters
 - can be independent of programming languages
- Compile the interface using IDL-compiler, resulting in
 - client stub (proxy)
 - server stub
 - auxiliary files (header files, ...)
- Client stub (proxy)
 - compiled and linked with client program
 - client program invokes remote procedure by invoking the (local) client stub
 - implements everything to interact with the server remotely
- Server stub
 - implements the server portion of the invocation
 - compiled and linked with server code
 - calls the actual procedure implemented at the server



Binding in RPC

- Before performing RPC, the client must first locate and *bind* to the server
 - create/obtain an (environment-specific) *handle* to the server
 - encapsulates information such as IP address, port number, Ethernet address, ...
- Static binding
 - handle is "hard-coded" into the client stub at compile-time
 - advantages: simple and efficient
 - disadvantages: client and server are tightly coupled
 - server location change requires recompilation
 - dynamic load balancing across multiple (redundant) servers is not possible
- Dynamic binding
 - utilizes a name and directory service
 - based on logical names, signatures of procedures
 - server registers available procedure with the N&D server
 - client asks for server handle, uses it to perform RPC
 - requires lookup protocol/API
 - may be performed inside the client stub (automatic binding) or outside
 - opportunities for load balancing, more sophisticated selection (traders)
- *Location transparency* usually means that a remote procedure is invoked just like a local procedure
 - Binding process for remote and local procedures usually differ



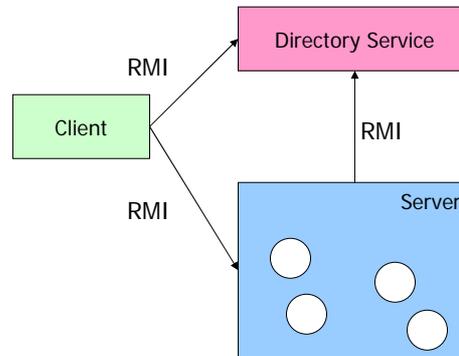
Variation 1: Distributed Objects

- Basic Idea: Evolve RPC concept for objects
 - application consists of distributed object components
 - object services are invoked using Remote Method Invocation (RMI)
- Utilizes/matches advantages of object-oriented computing
 - object identity
 - encapsulation: object manipulated only through methods
 - inheritance, polymorphism
 - interface vs. implementation
 - reusability

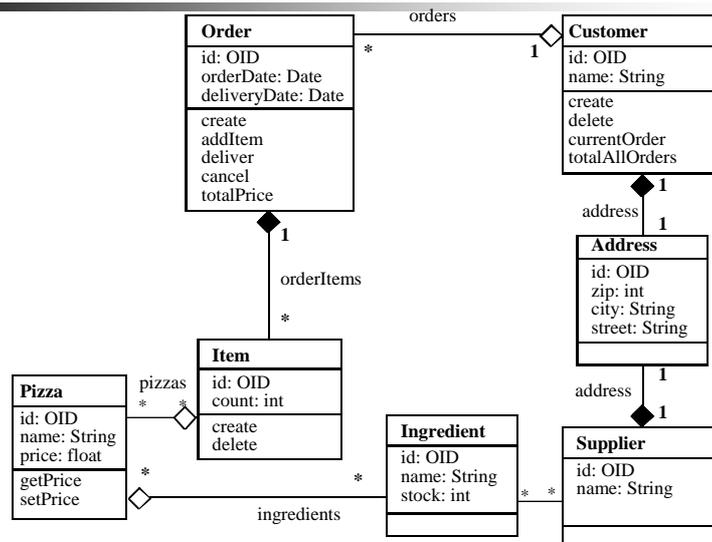


Distributed Objects with Java RMI

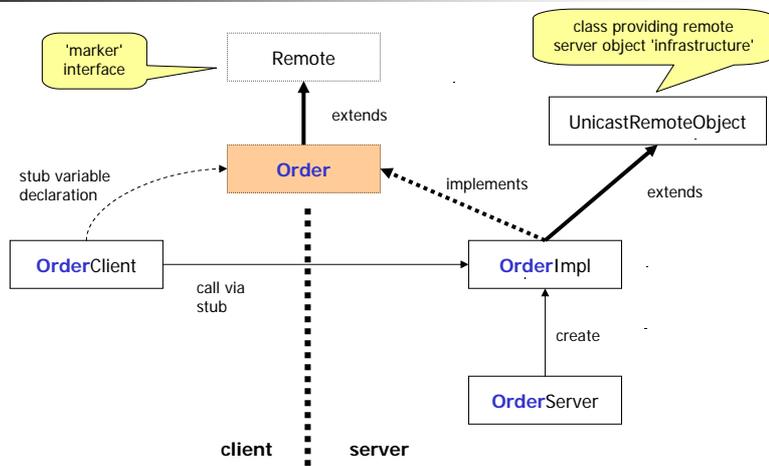
- Mechanism for communication
 - between Java programs
 - between Java programs and applets
 - running in different JVMs, possibly on different nodes
- Capabilities
 - finding remote objects
 - transparent communication with remote objects
 - loading byte code for remote objects



Example Scenario: Pizza-Service



Example - Class and Interface Relationships



Example – Remote Service Interface

```

import java.rmi.*;
import java.util.Date;
public interface Order extends Remote {
    public void addItem(int pizzaId, int number)
        throws RemoteException;
    public Date getDeliveryDate() throws
        RemoteException;
    public Date setDeliveryDate (Date newDate) throws
        RemoteException;
    ...
}
  
```

```

...
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.*;
  
```



Example – Server Class Implementation

```
...
public class OrderImpl
    extends UnicastRemoteObject
    implements Order {
    private Vector fItems;
    private Date fDeliveryDate;
    public OrderImpl(String name) throws RemoteException {
        super();
        try {
            Naming.rebind(name, this);
            fItems = new Vector();
            fDeliveryDate = null;
        }
        catch (Exception e) {
            System.err.println("Output: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
...
```

'export' Order
object for accepting
requests

register
with name
server



Example – Server Class (continued)

```
...
public void addItem(int pizzaId, int number )
    throws RemoteException {
    // assuming class Item is known
    Item item = new Item(pizzaId, number);
    fItems.addElement(item);
    }
... // Impl. of other methods    }
```



Example – Server

```
...
import java.rmi.*;
import java.server.*;
public class OrderServer {
    public static void main(String args[]) {
        try {
            OrderImpl order = new OrderImpl("my_order");
            System.out.println("Order server is running");
        }
        catch (Exception e) {
            System.err.println("Exception: " + e.getMessage());
            e.printStackTrace();
        }
    }
}
```

remote object
name (later used
in client lookup)



Example – Client Program

```
...
import java.rmi.*;
public class OrderClient {
    public static void Main(String args[]) {
        try {
            Order order = (Order)
                Naming.lookup("/my_order");
            int pizzaId = Integer.parseInt(args[0]);
            int number = Integer.parseInt(args[1]);
            order.addItem(pizzaId, number);
        }
        catch (Exception e) {
            System.err.println("system error: " + e);
        }
    }
}
```

returns an instance of the stub
class (generated from the remote
Order interface)



Example – Compile, Generate Stub, Run

- Compile:
javac Order.java OrderImpl.java OrderClient.java OrderServer.java
- Generate stub and skeleton code:
rmic OrderImpl
- Administrative steps:
 - Start directory server: *rmiregistry*
 - Start RMI-Servers: *java OrderServer*
 - Run clients: *java OrderClient*



Variation 2: Stored Procedures

- Named persistent code to be invoked in SQL, executed by the DBMS
 - SQL *CALL* statement
 - RPC is not transparent!
- Created directly in a schema or in a SQL-server module
- Have a header and a body
 - Header consists of a name and a (possibly empty) list of parameters.
 - may specify parameter mode: IN, OUT, INOUT
- SQL routines
 - Both header and body specified in SQL
- External routines
 - Header specified in SQL
 - Bodies written in a host programming language
 - May contain SQL by embedding SQL statements in host language programs or using CLI

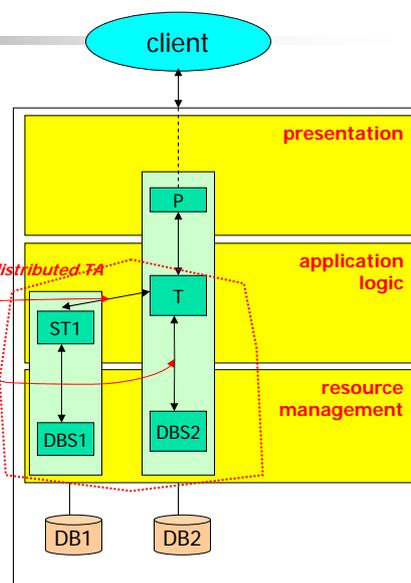


RPCs and Transactions

- Example scenario for T: debit/credit
 - T invokes debit procedure (ST1), modifying DB1
 - T performs credit operation on DBS2, modifying DB2
- Need transactional guarantees for T
- Program structure of T


```

BOT
CALL debit( ...)
CONNECT (DB2)
UPDATE ACCOUNTS SET ...
DISCONNECT
EOT
            
```
- Requires coordination of distributed transaction
 - based on 2PC



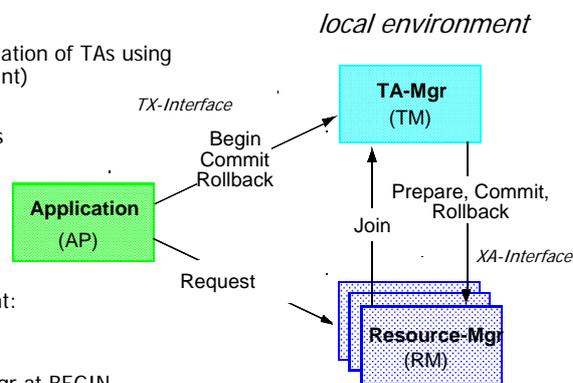
Transactional RPC (TRPC)

- Servers are resource managers
- RPCs are issued in the context of a transaction
 - demarcation (BOT, EOT) usually happens on the client
- TRPC-Stub
 - like RPC-Stub
 - additional responsibilities for TA-oriented communication
- TRPC requires the following additional steps
 - binding of RPC to transactions using TRID
 - notifying TA-Mgr about RM-Calls if performed through RPC (register participant of TA)
 - binding processes to transactions: failures (crashes) resulting in process termination should be communicated to the TA-Mgr



X/OPEN – Standard for Distributed TA Processing

- Resource Manager
 - recoverable
 - supports external coordination of TAs using 2PC protocol (XA-compliant)
- TA-Mgr
 - coordinates, controls RMs
- Application Program
 - demarcates TA (TA-brackets)
 - invokes RM services
 - e.g., SQL-statements
 - in distributed environment: performs (T)RPCs
- Transactional Context
 - TRID generated by TA-Mgr at BEGIN
 - established at the client
 - passed along (transitively) with RM-requests, RPCs



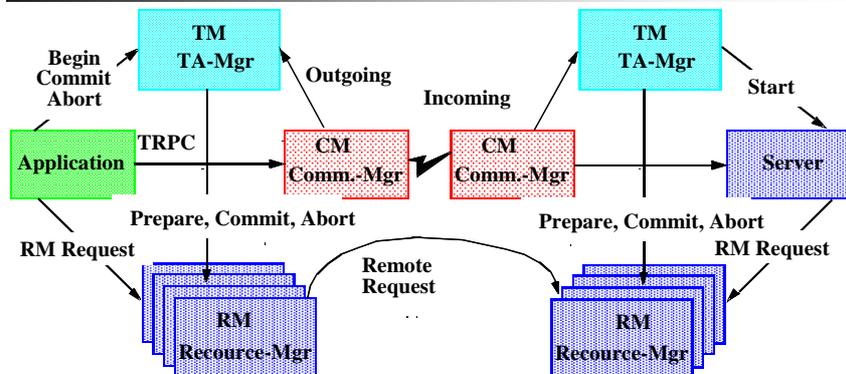
Interactions in a Local Environment

1. AP -> TM: begin() – establishes transaction context, global TRID
2. TM -> RM: start() – TM notifies frequently used RMs about the new global transaction, so that RM can associate future AP requests with the TRID
3. AP -> RM: request – the RM
 1. first registers with the TM to join the global transaction (unless it was already notified in (2) above), then
 2. processes the AP request
4. AP -> TM: commit() (or rollback) – TM will interact with RMs to complete the transaction using the 2PC protocol

A thread of control is associated with at most one TRID at a time. An AP request is implicitly associated with a TRID through the current thread.



X/OPEN DTP – Distributed Environment



- Outgoing TRPC: CM acts like a RM, notifies local (superior) TM that TA involves remote RMs
- Incoming TRPC: CM notifies local (subordinate) TM about incoming global TA
- Superior TM will drive hierarchical 2PC over remote TM/RMs through CM



© Prof. Dr.-Ing. Stefan DeBloch

23

Middleware for Heterogenous and Distributed Information Systems - WS06/07

Summary

- Remote Procedure Call
 - importance core concept for distributed IS
 - RPC model is based on
 - Interface definitions using IDL
 - Client stub (proxy), Server Stub for transparent invocation of remote procedure
 - Binding mechanism
- RPC Variations
 - Remote Method Invocation
 - supported in object-based middleware (e.g., CORBA, Enterprise Java)
 - Stored Procedures
- Transaction support for RPCs
 - distributed transaction processing guarantees atomicity of global TA
 - transactional RPC
 - X/Open DTP as foundation for standardized DTP
 - variations/enhancements appear in object-based middleware (CORBA OTS, Java JTA/JTS)



© Prof. Dr.-Ing. Stefan DeBloch

24

Middleware for Heterogenous and Distributed Information Systems - WS06/07