Distributed Objects (revisited)

- From three-tier and multi-tier client/server architectures to component-based architectures with distributed objects:

Three-tier / Multi-tier Systems
- Vertical distribution of layers

Component-based Systems
- Additional horizontal distribution
- Re-use of existing components
Distributed Objects (revisited)

- With distributed objects, a distinction is necessary between the development of components and the development with components.

  - Development of components:
    - Focus exclusively on the development of the functionality (application logic) of single components
    - Make services available via published interfaces

  - Development with components („Programming in the Large“)
    - Development of systems = consistent composition and integration of „off-the-shelf“ components
    - Re-usable software entities

  - Essential for information systems is the latter aspect, i.e., the development with components.

  - This requires appropriate infrastructure support for the composition task
    → Distributed Object Management

Terminology

- Object
  - Application object as abstract data type, has
    - interface (methods and attributes)
    - Object reference (unique, internal)
  - Design alternatives: state is
    - explicit: state of the persistent data from the database are materialized explicitly in the form of attributes of the object
    - implicit: state is not materialized in the object but is made available by methods that access the database

- Client
  - Software entity that call methods of some object

- Server
  - Software entity that provides some object implementations

Hence, objects can both play the role of clients and the role of servers
**RPC versus Object Bus**

**Client**
- **RPC**
  - Call `foo`

**Server**
- **RPC Mechanism**
  - Execution of the same procedure / function
  - Independent of client call

**Object Bus (ORB)**
- **Client**
  - Invoke `foo` on Object X
  - Invoke `foo` on Object Y
- **Server**
  - Execution of a method within a remote object
  - Different invocations might react differently (due to polymorphisms, etc.)

---

**8.1 DOM Requirements**

- **Location transparency**: Call of server method has to be independent of the physical location of both server and client object
  - Possibility to have method calls across process boundaries
- **Platform independence**: Hardware and operating system independence for both client and server objects, i.e., co-existence of several platforms
  - Interoperability
- **Language independence**: Embedding into existing programming languages, but: Client and server implementations shall be independent of a particular programming environment
  - Separation of interface and implementation
- **Static and dynamic method calls**
- **transactions**
- **Self-description**
- **Polymorphism**
- **Security**
- **Support of “legacy” systems**
Middleware – Realization at a Glance

- What characteristics does a middleware infrastructure have to support distributed calls in heterogeneous environments?
- Location transparency?
- Platform independence?
- Language independence?

Java RMI — An Overview

- RMI: Remote Method Invocation
  - Call of methods in remote VMs
  - Server object uses proxy (client stub) as "representative" in the client VM
    - Client Stub is loaded at runtime into the client VM and
    - Realizes the Marshaling of requests
  - Invocations go to a server skeleton in the remote VM
    - Un-marshaling
    - Method call to the server object
  - Registry as directory service for remote calls
Java RMI — Application

Client Development
- Implement client code „MyClient“ (use `java.rmi.Naming` to locate remote objects; call remote object methods via client stubs)
- `javac myClient.class`

Server Development
- Define remote interface of „MyServer“ (extend `java.rmi.Remote` Interface)
- Implement remote-Interface of „MyServer“ (derive Java server class from `java.rmi.UnicastRemoteObject`)
- `javac myServer.class`
- `rmic` (Stub-Compiler)
- `Server Skeleton (.class)`
- `myServer.class`
- `Server Class`
- `Client Stub (.class)`
- `Client Registry`
- Start RMI Registry
- Instantiate server objects
- Register instance of server object (`java.rmi.Naming`)

Java RMI — Summary
- Location transparency
  - Method calls in remote VMs possible
  - RMI Registry as Naming Service
- Platform independence
  - Is provided, since just a Java VM is required for client and server
- Language independence
  - Limitation to Java, no other language binding
  - No dynamic method calls
  - No transaction (without extensions)
8.2 CORBA — An Overview

- Standard of the OMG (= Object Management Group), www.omg.org
  - One of the world’s largest software consortia, consisting of approx. 800 software companies (e.g., Sun, IBM, HP, Oracle, Microsoft)

- CORBA = Common Object Request Broker Architecture
  - Open interoperability standard for heterogeneous, distributed, object-oriented systems
  - First version has been published already in 1991 (that means that the standard has been defined before products, i.e., CORBA implementations, have been on the market)

CORBA Middleware at a Glance

- Main component is the ORB (Object Request Broker), CORBA’s object bus
  - Common communication platform for distributed objects
  - Interaction between distributed objects again via local proxies of remote objects (Client IDL Stubs), which make the actual method calls via server skeletons
    - Each remote call appears to the client as a local call
    - Client IDL stubs hide the language in which the server object is implemented

- Diagram showing the components of CORBA middleware:
  - Client Objects
  - Horizontal Facilities
  - Vertical Facilities
  - Common Object Services
  - Server Objects
  - Object Request Broker
  - Operating System and Network Hardware
Components of CORBA

- CORBA consists of
  - its interface definition language IDL and corresponding language bindings (e.g., C / C++, Java, SmallTalk, Ada, Lisp, Cobol)
  - a common control infrastructure, the Object Request Broker (ORB),
  - a protocol that guarantees interoperability between different ORB implementations: the Internet Inter-ORB Protocol (IIOP),
  - the specification of common services (the so-called Common Object Services, COS) and the Common Facilities and Domains.

Interface Definition Language (IDL)

- Language for defining object interfaces (only interfaces, no implementation)
  - independent of a concrete programming language / system
  - influenced by the syntax of C++
    - Additional constructs: exceptions, strings, sequences, ...
    - but without: pointers, templates, all kinds of control structures, ...
  - IDL definition is mapped by a pre-compiler to a programming language which will be used for the implementation
    - Ada, C, C++, COBOL, Java, Lisp, or Smalltalk
  - IDL more or less defines the „schema“ of a distributed application

- Object interfaces
  - Attributes, operations, exception handlers
IDL: An Example

```java
// naming service constants
const string cs341ContextName = "cs341_hs2014";

module Bank
{
    typedef long AccountID;
    typedef float Money;

    interface Account
    {
        readonly attribute AccountID number; // account id
        readonly attribute Money balance; // amount of money in account

        // thrown if not enough money on account to withdraw
        exception InsufficientBalance {};

        // public methods to modify an account
        void Credit ( in Money amount );
        void Debit ( in Money amount ) raises (InsufficientBalance);
    }
};
```

Object Request Broker (ORB)

- Backbone of CORBA:
  - the communication kernel allows objects to communicate via process and network boundaries (object bus)
  - Provides basic mechanism to execute methods of remote objects
  - Acts as a broker that forwards the method calls of a client to a server with the proper implementation (and starts this server, if needed)
  - Marshalling / De-marshalling

- Client
  - Contains only proxy objects and knows their interfaces (because of IDL)
  - Is neither aware of the location not of the language of the implementation
  - Sends request (remote destination object, operation, parameters, ...) to the broker

- Server
  - Contains the object implementation (data and method code)
How the ORB Works

- Localize the object implementation
- Prepares server for the method call and execution (starts server process, if necessary)
- Transfers method and parameters to the server / transfers the results back to the client
- Converts data between different hardware architectures, if needed
- Provides exceptions as error handling mechanism

Internet Inter-ORB Protocol (IIOP)

- The Internet Inter-ORB Protocol defines the communication between different ORBs
  - Transfer and message format: GIOP
  - Interoperable Object References: IOR
  - Transport protocol: TCP/IP
### Common Object Services

- Common Object Services extend and complete ORB functionality
  - Define basic services for object management
  - Are independent of a concrete application
- Naming Service
  - Allows objects to bind to / find other objects with meaningful names
- Event Service
  - Asynchronous communication between objects via events
- Security Service
  - Authentication, access control, secure communication, etc.
- Transaction Service
  - Coordinator for transactions on top of distributed objects
- Persistent Object Service
  - Store objects persistently in different servers

### … Common Object Services

- Query Service
  - Declarative access to objects, e.g., via SQL-92 or OQL
- Trader Service
  - "Yellow Pages" of a CORBA system
- Externalization Service
  - Allows to store objects in a data stream
- Licensing Service
  - Mechanism to control licenses by software vendors
- Time Service
  - Time synchronization in a distributed system
- Concurrency Control Service
  - Implements a lock manager for locking objects
- …
Common Facilities

Highest level of abstraction in the CORBA architecture

• Horizontal common facilities (application independent)
  - Internationalization & Time
  - Mobile Agent Facility
  - ... (User Interface, System Management, etc.)

• Vertical market facilities – CORBA Domain Technologies (application dependent „business“ objects)
  - CORBAfinance
  - CORBAmanufacturing
  - CORBAecommerce
  - CORBATelecom
  - CORBAmued
  - CORBAttransport

The Complete Anatomy of the ORB

Client
• IDL Stubs: static, i.e., all operations are known at compile time
• Dynamic Invocation Interface (DII) finds objects and interfaces at run-time and constructs the call dynamically, by using the
• Interface Repository (for interpreting tools like the Query Browser)

Server
• Object Adapter interprets object references, maps these to object implementations and calls method implementations
• Statically via IDL Skeletons or
• Dynamically via Dynamic IDL Skeletons as basis for the implementation of application logic
CORBA — Summary

- Location transparency
- Platform independence
  - Basic functionality of the ORB; additionally:
  - IIOP (Internet Inter-Orb Protocol) allows for the communication between different ORBs
- Language independence
  - Strict separation of interface and implementation
  - Language-independent definition of interfaces by IDL
    (but this IDL is NOT identical to other IDLs, e.g. the one of COM+)
  - Bindings to several programming languages exist
    (C, C++, Smalltalk, Ada, Java, PL/1, ...)

- Supports both static and dynamic method calls
- OTS: method calls can be embedded into transactional contexts

Summary of Major Middleware-Frameworks

<table>
<thead>
<tr>
<th></th>
<th>Java RMI (Remote Method Invocation)</th>
<th>CORBA (Common Object Request Broker Architecture)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Location transparency</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Platform independence</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Language independence</td>
<td>Java Clients or Servers</td>
<td>✔</td>
</tr>
</tbody>
</table>

Current Trends:
- Middleware-Frameworks come closer together (or even smoothly melt):
  Enterprise JavaBeans (EJB) support both Java RMI and CORBA
- Individual usage of components:
  application-specific configuration of components („deployment“), independent of the composition of components
Brave New DOM World?

- Location transparency sounds great, but is worth asking how far (and with which price) it can be actually achieved …
- Some problems that are often overlooked or even ignored:
  - Latency: access to remote objects is in orders of magnitude slower than local access
    → may lead to low performance of distributed applications
  - Memory Access: pointers to local data are not valid
    → common source of failures
  - Partial Failure: instead of complete programs, only parts can fail
    → debugging of distributed applications may be a pain!
  - Concurrency: distributed objects have to support concurrent calls
    → consequences for transaction management

- Yet, the true challenge is a good design of a distributed information system!

8.3 Object Transaction Service

- Object Transaction Service (OTS)
  - Is part of the CORBA Common Object Services (COS Transactions)
  - Object-oriented equivalent of the X/Open DTP standard
    → Uses XA interface of the underlying databases
- OTS specifies interfaces (via IDL) and functionality of a transaction service
- Basic idea: a transaction context is assigned to each transaction
  - Method calls are associated with this context
    → Several different requests can be assigned to a single transaction
  - OTS equivalent to the transaction ID (TXID) which takes over this role as part of the TxRPC
OTS: Supported Transaction Models

- Flat Transactions
  - Transaction bracket by
    ```
    begin transaction
    ...
    end transaction
    ```
  - Everything within this transaction bracket is executed with ACID guarantees (coordination by 2PC protocol; usage of XA interface of the underlying databases)
  - Support in OTS is mandatory

- Nested Transactions
  - Main idea: independent sub-transactions within a transaction
    - More flexible failure handling
  - Support is optional (and if provided, only as closed-nested transactions)

Closed Nested Transactions in the OTS

- Sub-transactions can be (tentatively) completed with `commit`, but they are isolated from parallel sub-transactions
- The commit of all sub-transactions does not take place before the commit of their top-level transaction
- In case of a rollback, ALL sub-transactions are rolled back (even when they have already [tentatively] committed)
  - All successful sub-transactions are completely executed in the sphere of control of their top-level transaction
- But: the commit of the top-level transaction is even possible when single sub-transactions have to be rolled back. Hence, failure handling by using alternatives is possible (e.g., $T_{1112}$ as alternative to $T_{1111}$)
OTS at a Glance

Transaction Context via OTS

- Transaction context explicitly set via `CosTransactions::TransactionFactory`
  - Returns `Control` object that represents the transaction
  - This object can be forwarded explicitly with a method call

- Transaction context can be implicitly set via `Current` object. This is a pseudo object that represents the transaction
  - The `Current` object implicitly assigns each thread a transaction context
  - Together with each method call, the ORB propagates this context to all transactional objects, i.e., to those that are derived from `CosTransactions::TransactionalObject` (these are either transactional server or recoverable server)
    - Propagation can span several call hierarchies
    - The transaction context is copied each time by the object adapter
  - Start and termination of transactions (commit / rollback) supported
  - Object needs to be derived from `Current`

- In both cases, the management and processing of the 2PC protocol is done automatically by the `CosTransactions::Coordinator`
IDL of the OTS - General Declarations

Status of a transaction in OTS:

```cpp
class Status {
    StatusActive,
    StatusMarkedRollback,
    StatusCommitted,
    StatusPrepared,
    StatusRolledBack,
    StatusUnknown,
    StatusNoTransaction,
    StatusPreparing,
    StatusCommitting,
    StatusRollingBack
};
```

2PC decision:

```cpp
class Vote {
    VoteCommit,
    VoteRollback,
    VoteReadOnly
};
```

Pre-defined exceptions (excerpt):

```cpp
exception NoTransaction {};
exception NotPrepared {};
exception Inactive {};
...```

---

IDL of the OTS (2) - TransactionFactory & Control

- **TransactionFactory**
  - Allows a client to start a new transaction (in case of nested transactions: creation of a new top-level transaction);
  - for this, the client needs to be derived from `CosTransactions::TransactionFactory`

```cpp
interface TransactionFactory {
    Control create(in unsigned long time_out);
    /* creates new top-level transaction and returns
       the corresponding control object */
...
};
```

- `create` generates a control object (used to identify a new transaction)

```cpp
interface Control {
    Terminator get_terminator ();
    Coordinator get_control ();
};
```
IDL of the OTS (3) - Terminator & Coordinator

- **Terminator**: supports the finalization of transactions
  (is used in the same way than the initiator of a transaction)

```java
interface Terminator {
    void commit ( in boolean report_heuristics);
    void rollback ();
}
```

- **Coordinator**: operations that are used by the participants of a transaction
  (recoverable server), e.g., for the registration of resources

```java
interface Coordinator {
    RecoveryCoordinator register_resource ( in Resource r );
    Status get_status ();
    Status get_parent_status ();
    Status get_top_level_status ();
    ...
}
```

IDL of the OTS (4)

- OTS uses a 2PC protocol. The **Resource** interface contains all operations that
  are required and used by the OTS for transaction control

```java
interface Resource {
    Vote prepare ();
    void rollback ();
    void commit ();
}
```
Example: Transfer (Explicit Transaction Semantics)

```c++
void FinanceImpl::Transfer( Bank::AccountID a1, Bank::AccountID a2,
Bank::Money amount, CORBA::Environment &env)
{
    try {
        CosTransactions::Control c;
        CosTransactions::Terminator t;
        c = TFactory->create(0); // create transaction context
        t = c->getTerminator;

        // get object references to needed Account objects
        source = Bank::Account::_bind("1001@bank1:Bank_Account_cs341");
        destin = Bank::Account::_bind("3004@bank1:Bank_Account_cs341");

        // transfer money calling methods of Account objects
        source->Debit(amount, c, env);
        destin->Credit(amount, c, env);
        t.commit(true); // commit transaction
    } catch (...) {
        // indicate errors to caller by TransferFailed exception
        t.rollback();
        throw Bank::Finance::TransferFailed();
    }
}
```

IDL of OTS – Implicit Transfer of Transaction Context

- The `Current` interface specifies all operations a transactional client or a participant of a transaction can use, if the transaction context is propagated implicitly by the CORBA runtime environment (ORB)

```plaintext
interface Current
{
    void begin();
    void commit ( in boolean report_heuristics );
    void rollback();
    Status get_status();
    Control get_control();
    void set_timeout();
    ...;
};
```

- The `TransactionalObject` interface indicates that an object is transactional. This also means that it receives the transaction context in case it participates in a transaction that a client has initiated.

```plaintext
interface TransactionalObject { };
```
Sample IDL of a Resource with OTS

- Account objects shall automatically receive the current transaction context
- In addition, the resource that manages the explicit state of account supports the 2PC protocol (thus, account objects have to register themselves as resources)

```idl
interface Account : COSTransactions::TransactionalObject, COSTransactions::Resource
{
    readonly attribute AccountID number;
    readonly attribute float balance;

    exception InsufficientBalance {};

    void Debit (in float amount) raises (InsufficientBalance);
    void Credit (in float amount);
};
```

Code Example: Transfer() with OTS (Current)

```cpp
void FinanceImpl::Transfer(Bank::AccountID a1, Bank::AccountID a2,
                          Bank::Money amount, CORBA::Environment &env)
{
    try
    {
        Current.begin(); // start new transaction

        // get object references to needed Account objects
        source = Bank::Account::_bind("1001@bank1:Bank_Account_cs341");
        destin = Bank::Account::_bind("3004@bank1:Bank_Account_cs341");

        // transfer money calling methods of Account objects
        source->Debit(amount, env);
        destin->Credit(amount, env);

        Current.commit(true); // commit transaction
    }
    catch (....)
    {
        // indicate errors to caller by TransferFailed exception
        Current.rollback();
        throw Bank::Finance::TransferFailed();
    }
}
```
## Further Reading

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>Red 96</td>
<td>J.-P. Redlich: <em>CORBA 2.0 – Praktische Einführung für C++ und Java</em>. Addison-Wesley, 1996</td>
</tr>
</tbody>
</table>