

Department of Computer Science

PURDUE
UNIVERSITY

CS505: Distributed Systems

**Lecture 8: Programming Support
Introduction**

Issues

▶ Data interoperability

- Big vs. little endian
- Types more generally

▶ Failures

- Non-masking/handling: need to cater for exceptions
- Masking: possibly (software) architecture choices

▶ (Hardware) architecture/deployment

- Assumptions must be made
- Specific layout

Data Interoperability

- ▶ **Requires pre-defined interoperable formats, e.g.,**
 - XDR in SUN NSF/RPC
 - IIOP in CORBA
 - XML in WS

Two-tier Architectures

▶ Developed in 80's

- Descending from file servers

▶ Tiers

- *Server*

- Hosts data, application logic (fat server)

- *Client*

- Hosts GUI, application logic (fat client)

▶ A.k.a. client/server architecture

Three-tier Architectures

► Tiers

- **Data storage**
 - “Raw” data
- **Application server**
 - Main application logic
- **Clients**
 - Remote access, mainly representation

Approach “Levels” [Briot et al.’95]

1. Language

- + Safety, simplicity, expressiveness
- Interoperability, flexibility

2. Reflection

3. Middleware (“libraries”)

- + Interoperability, flexibility
- Safety, simplicity, expressiveness

Library Approach

▶ Group communication toolkits

- Isis [Birman et al.], ...
- In Java JGroups etc.

▶ Explicit way of going about replication

- Programmer deals with
 - Plurality (multiple replies)
 - State transfer: implement `get-/setState()`
 - Joining, leaving
 - Required guarantees: reliable, total order, causal order, ...

Example

```
public class MyServer extends Skeleton implements Replica {  
  
    public Object getState() { ... }  
    public void setState(Object state) { ... }  
  
    public MyServer() { super(); join("/myGroup"); }  
  
    public Object receive(Object[] args) { ... }  
    ...  
}
```

```
public class MyClient {  
    ...  
    public static void main(String[] args) {  
        Group g = Group.lookup("/myGroup");  
        g.setProtocol("tobcast");  
        Object[] rets = g.broadcast(...);  
        ...  
    }  
    ...  
}
```

Transactions

► Transactions can be similarly dealt with

```
public class MyTransactionalServer ... implements Transactional {  
  
    public boolean vote() { ... }  
    public void commit(...) { ... }  
    public void abort(...) { ... }  
    ...  
}
```

► And even

```
public class MyTransactionalServer ... implements Transactional {  
  
    public void myMethod1(myArg1 arg1, ...,  
                        TransactionContext ctxt) {...}  
    ...  
}
```

Evaluation

- ▶ **No surprises**
- ▶ **Servers/replicas must implement API**
 - For state transfer, commit etc.
- ▶ **Common approach: application server**
 - Components are replicated
 - Run under the control of container
 - Every in-/outgoing invocation is intercepted

Reflection Approach

- ▶ **Initial idea based on interceptors**
 - Filters for in-/outgoing invocations

- ▶ **Flown into aspect-oriented programming**
 - Replication aspect
 - Transaction aspect
 - ...

Example

```
public class Teller {
    public void transfer(BankAccount account1,
                        BankAccount account2) {
        my1stAccount.withdraw(amount);
        my2ndAccount.deposit(amount);
    }
}
```

```
aspect Transactionizer {
    pointcut transfer() :
        calls(void Teller.transfer(BankAccount, BankAccount));
    around() : transfer() {
        boolean aborted = false;
        Transaction t = new Transaction();
        try { t.start(); proceed(); }
        catch(TransactionException e) {
            aborted = true;
            t.abort();
            throw e;
        } finally { if (!aborted) t.commit(); }
    }
}
```

Evaluation

▶ **Known benefits of AOP/reflection approach**

- **Separation of concerns**
 - **Divide efforts and expertise among programmers**

▶ **Known disadvantages**

- **Transparency in main code misleading**
 - **Falsely promotes orthogonality**
- **Same reason why failures can not be hidden, masking can not be**
 - **Enough hints that full separation impossible and not reasonable**

Integration Approach

► Transactions

- Need to denote “atomic” sections (e.g., methods)
- What to do in case of abort?
 - Client-side: retry automatically or exception
 - Server-side: automatic rollback?

► Replication

- Replicated objects are invoked as usual through proxies
- Multiple replies and policies handled through libraries

Example

```
public class MyBankServer {  
  
    public void withdraw(Account acc, int amount) {... }  
    public void deposit(Account acc, int amount) {... }  
    ...  
  
}
```

```
public class MyBankClient {  
    private MyBankServer server;  
  
    public void transfer(Account from, Account to, int amount) {  
        atomic {  
            server.withdraw(from, amount);  
            server.deposit(to, amount);  
        }  
        ...  
    }  
    ...  
  
}
```

Evaluation

▶ **Clearly cleanest approach**

- **Division of responsibility**
 - without false promises

▶ **Drawback: interoperability**

- **Not all languages support it**
 - Somebody has to lead the path...