Introduction to Enterprise Computing

Giovanni Chierico
CERN (IT-AIS-HR)

Inverted CERN School of Computing

Presentation “prerequisites”

The presentation doesn’t go into too much details, but it might be useful to have:

- General knowledge of distributed systems
- Some experience with OO Programming
- Some Java Experience

Presentation Overview

- What is “Enterprise Computing”
- Common Problems
- Real World Solutions
- Common Patterns
  - Naming Services
  - Pooling
  - Transaction Management

What is Enterprise Computing

What is “Enterprise Computing”

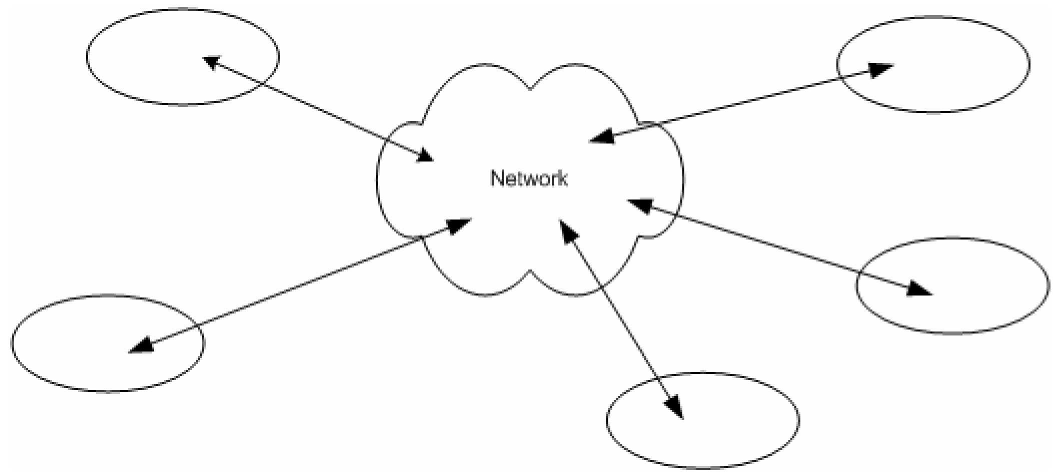
Solving computing problems in a

- Distributed
- Multi-tier
- Server-centric environment.

Common in big companies (like CERN) where users access a variety of applications that share data and resources, often integrated with legacy systems.

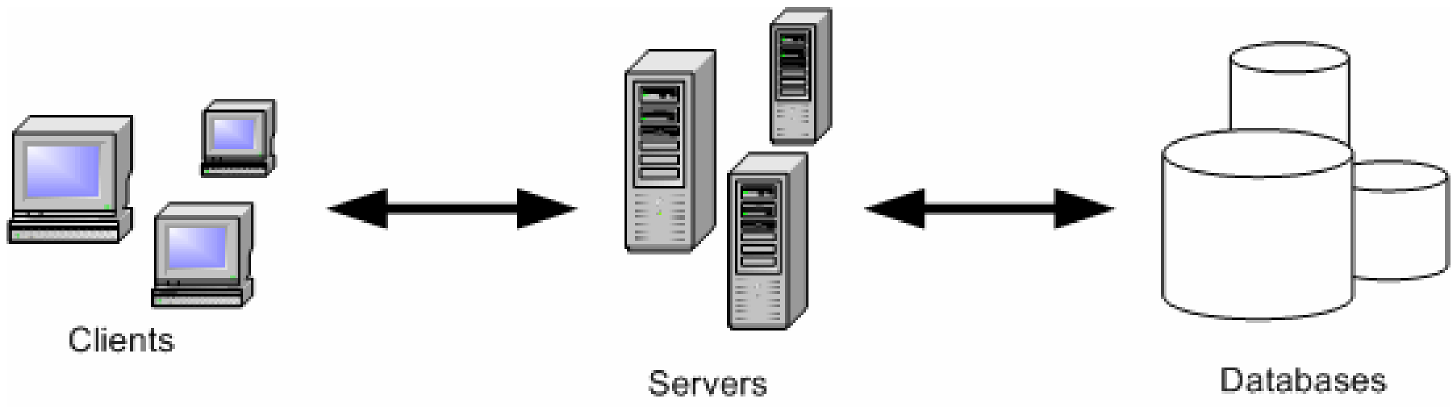
Distributed

- Means that the *components* that make up our system could be living on different machines and communicate through the network
- Components must be able to find each other and to communicate effectively



Multi-tier

- Many distributed schemas are possible (e.g. P2P)
- In an enterprise environment we can identify components having very different roles (client, server, database) and different requirements



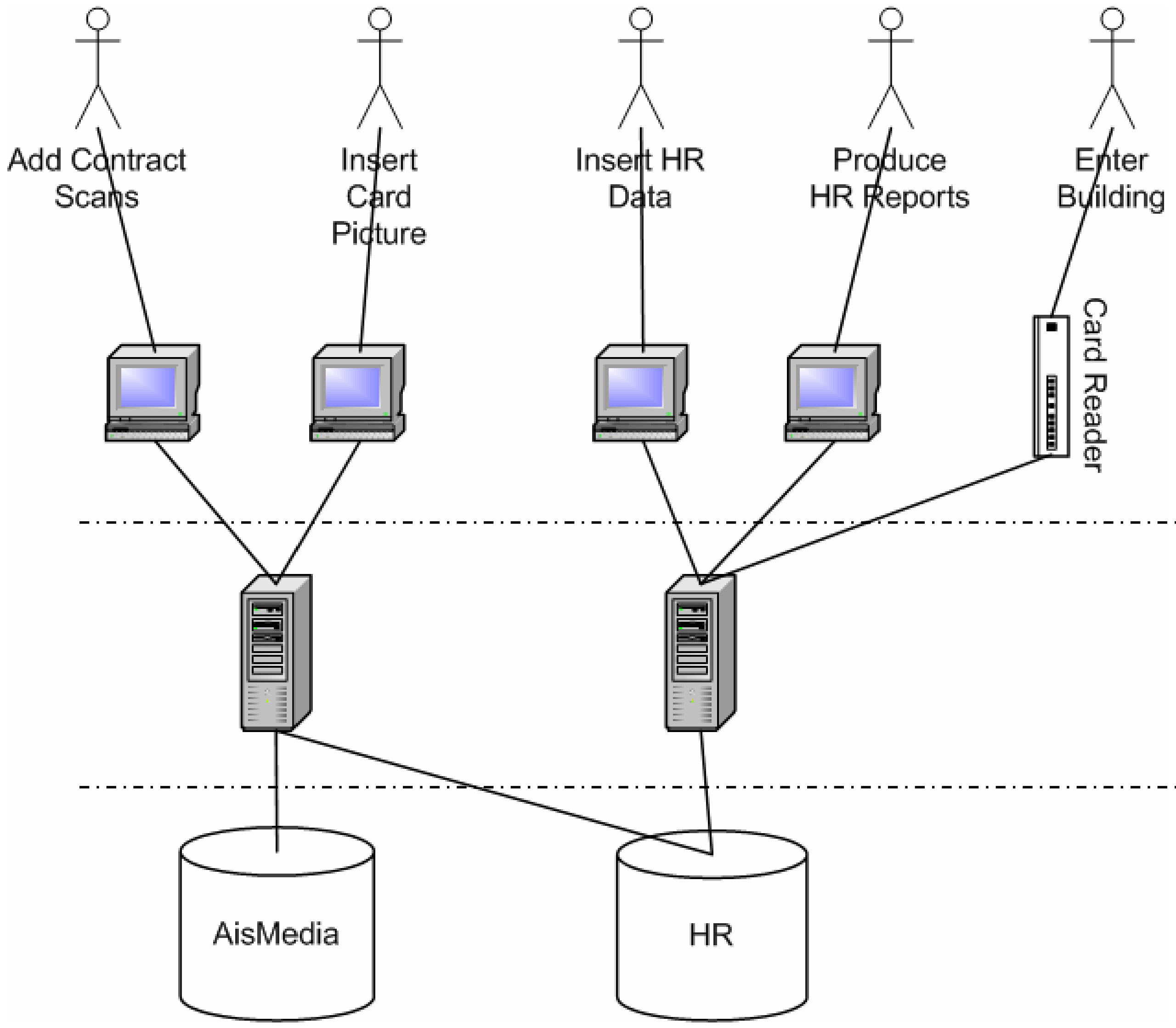
Server centric

- The server performs the operations needed to solve our particular problem (*business logic*).
- We are not going to write a new DB or a new client technology. They have to be as much *standard* as possible to minimize technology dependencies and deployment problems.

Common 3-tier architecture

- Client
  - Interfaces with the user
- Server
  - *Implements Business Logic*
  - Implements Middleware
- Database
  - Persistently stores data
  - Retrieves stored data

Example



Client

Application Server

Database

Common Problems

Common Problems/Services (I)

- Remote method invocation
  - Logic that connects a client and a server via a network connection: dispatching method requests, serializing data, …
- Load balancing
  - Client must be directed to server with lighest load
- Transparent fail-over
  - If server crashes the client should be redirected to another one without interruption of service
- System integration
  - New components must be integrated into existing system: different languages, OSs, hardware, …
- Transactions management
  - Operations must be reliable (ACID) even across different machines and networks

Common Problems/Services (II)

- Logging
  - If something goes wrong we need a log to reconstruct the events and indentify the problem
- Threading
  - If many clients connect to the same server we must be able to process the requests simultaneously
- Messaging
  - If components are loosely coupled we need to manage messages, queues, consumers, producers
- Pooling
  - We must be able to pool and reuse expensive resource
- Caching
  - We should be able to cache our data (e.g. DB query results) at different levels to improve performance

Middleware

- All these services together can be called *middleware* because they don’t implement our *business logic*, but yet they have to be present in our system
- Should be present in the *framework* we use
- Should be more *configured* than programmed

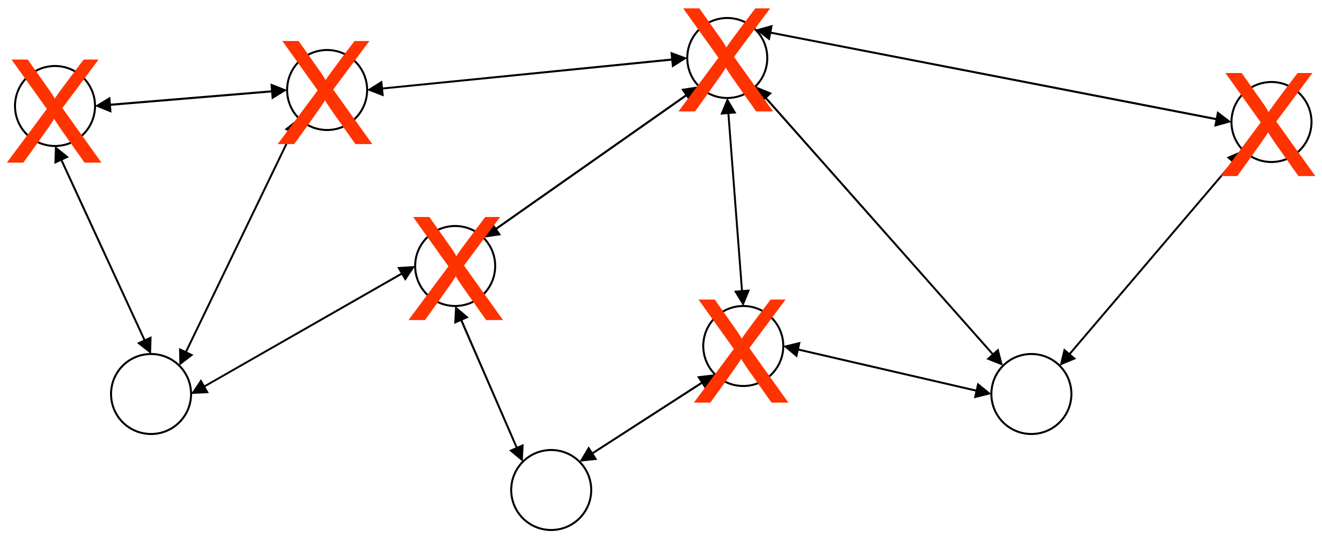
CERN School of Computing

Complexity

- System complexity is the #1 problem in enterprise systems
  - New functionalities are added to existing components
  - New components are created and must be integrated with existing ones
  - New technologies are introduced
  - Old technologies are phased out
- Quality of service has to be maintained
- Developers come and go
- Software ecosystems

Dependencies

- Dependencies are evil!
  - Resilience
  - Regressions

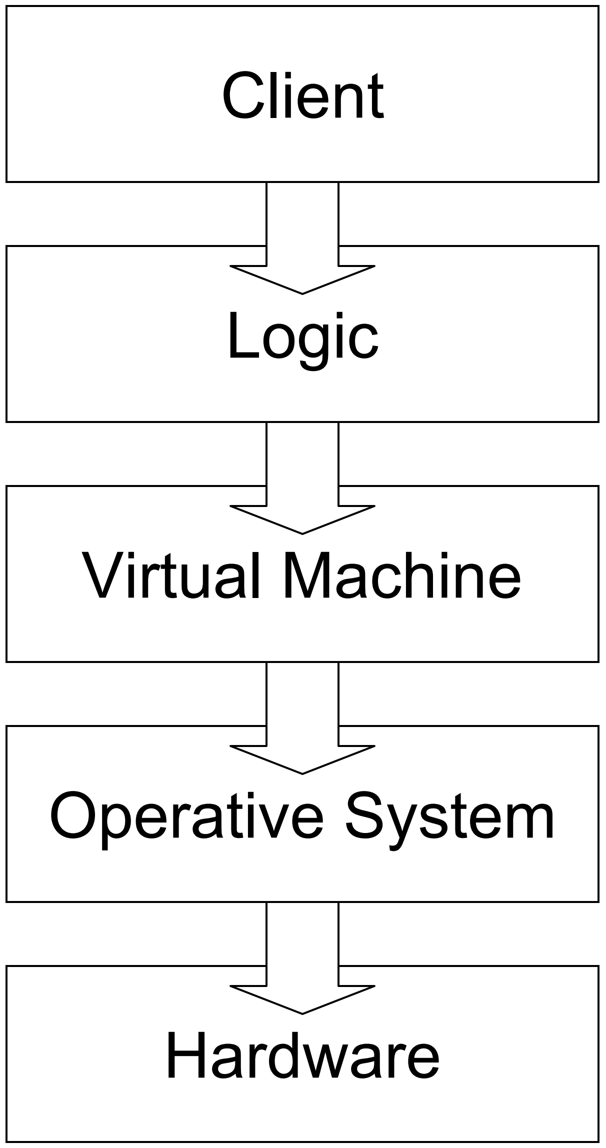


CERN School of Computing

Abstraction

- Successful systems provide *abstraction layers* to limit dependencies
  - Don’t need to understand internal combustion to drive a car
  - Don’t need to understand TCP/IP to browse the web
  - Don’t need to signal modulation to use your cell phone
- *You shouldn’t care* about the underlying abstraction layers.
- Unless there are problems!!!
- When you need another abstraction layer, use a *standard* one

Software Abstraction Layers



- If you write the Logic you shouldn't care about
  - How the virtual machine opens a file
  - How the OS stores the file
  - How the hard disk organizes the sectors
- Trade-off flexibility vs complexity

Optimization

“Premature optimisation is the root of all evil in programming”

Donald E. Knuth[1]

- If the system is well designed, functionalities and performance are indipendent.
- Optimization usually creates dependencies
- Relying on particular implementations (non-standard public features) is dangerous
- Relying on particular “internal” features (e.g. Oracle SQL-optimization) is a sign something is going wrong

1) Professor Emeritus of “The Art of Computer Programming” at Stanford University

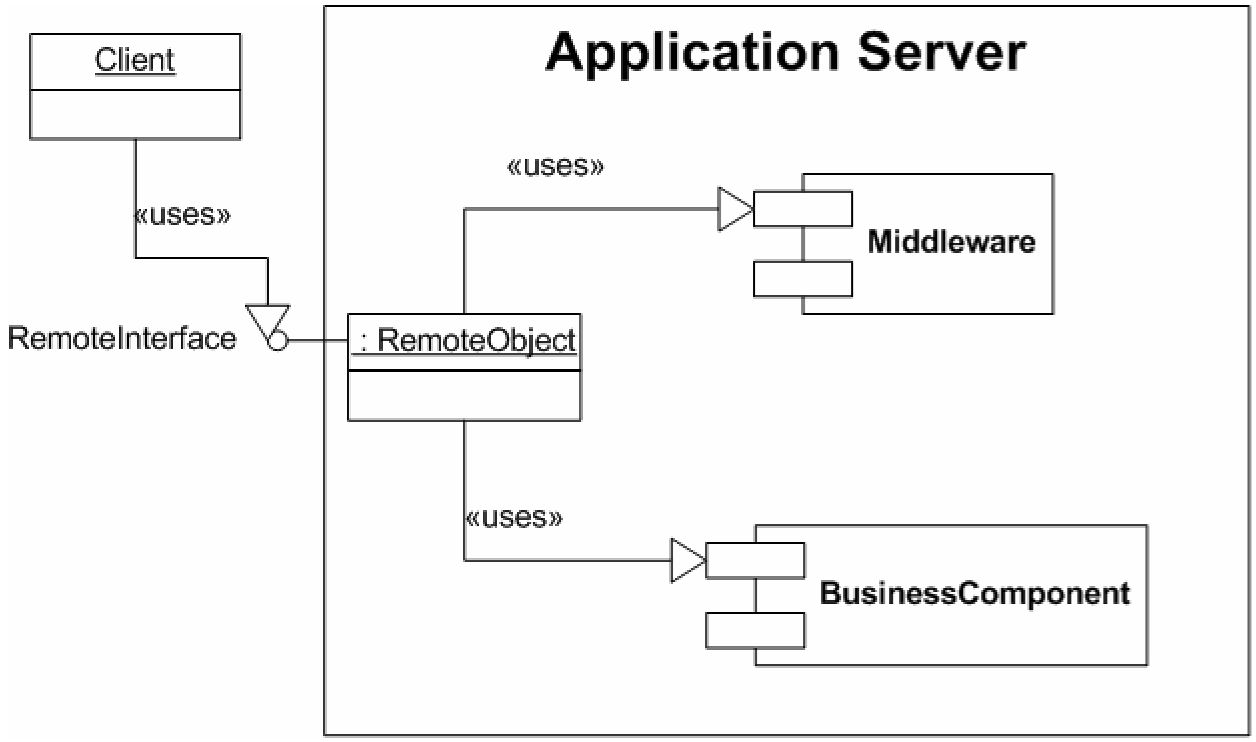
Programming Vs Configuring

- Easier to write/understand/maintain/deploy a configuration file than some program code
  - Less dependend on the implementation
  - Semantics are more evident
- *If well designed* it gives you
  - All the functionalities you need
  - Enforces good design
  - Makes it more difficult to “shoot yourself in the foot”

See inversion of control pattern and the Spring Framework in following lectures

Real World Solutions

Application Server



•Client uses remote interface
•Remote Object is managed by Application Server
•Transparent use of middleware
•Reduced dependencies

Java Enterprise: J2EE

Java 2 Enterprise Edition

Sun Microsystems publishes J2EE specifications telling what services an Application Server must provide.

Specifications are defined with the Java Community Process, with the partecipation of many leading companies and individuals.

A whole set of specifications are available for anyone to implement (JAXP, JMS, JNDI, JTA, JSP, JDBC, ...) and new ones are always proposed and evaluated (Java Specification Requests)

J2EE Vendors

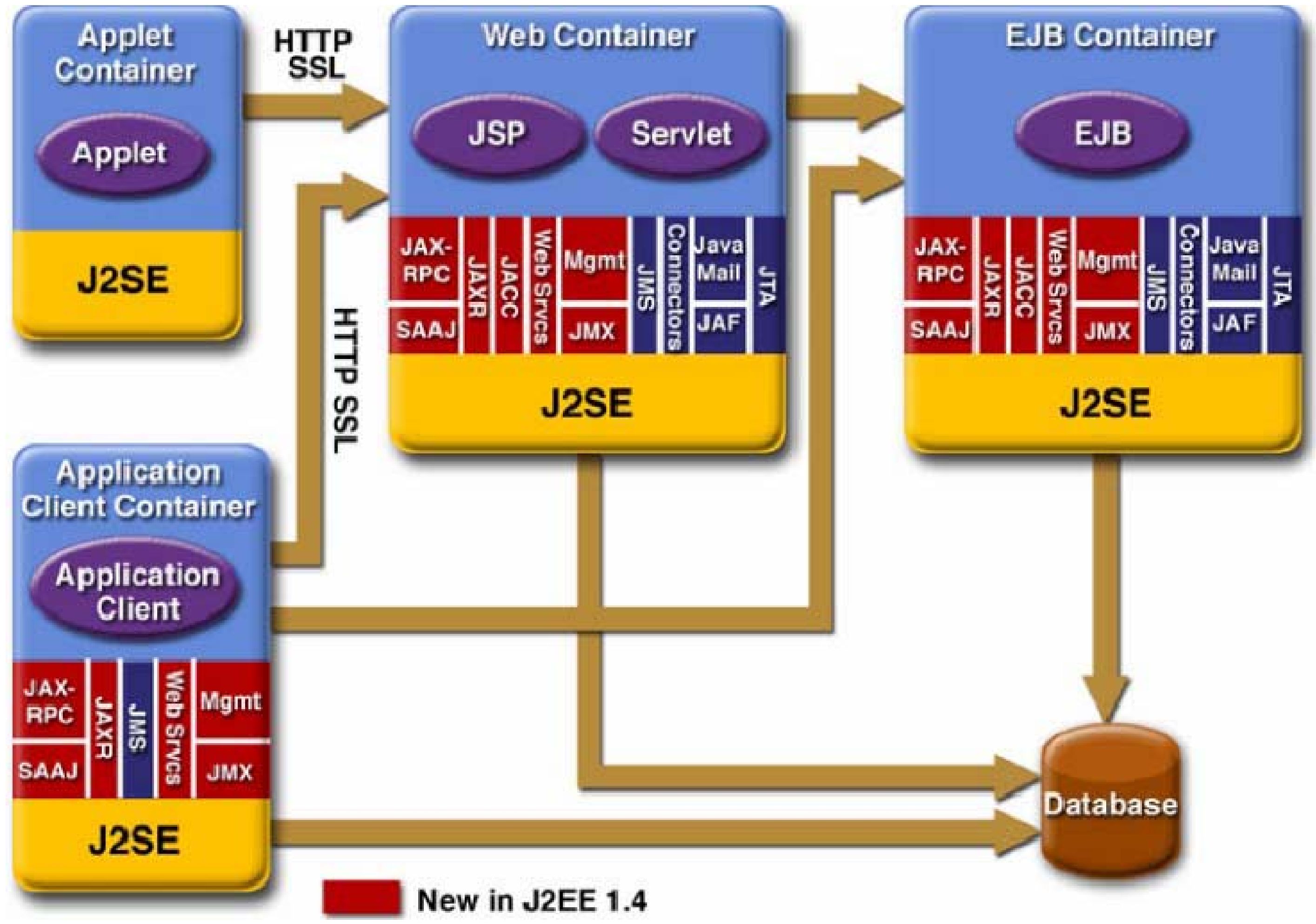
Various vendors (IBM, BEA, Oracle, JBoss) implement the J2EE specifications, competing in the Application Server market.

The implementations are tested by Sun that certifies the Application Servers as J2EE compatible. This guarantees that an application written following *strictly* the J2EE standards will run unmodified under the various implementations.

Vendors compete on the implementation quality (performance), and adding non-standard features.

CERN School of Computing

J2EE stack



Microsoft .NET

Similar services are provided by the .NET platform.

Of course there's no one-to-one strict correspondence…

And no real competition

MS.NET	J2EE
ASP	JSP/JSF
DCOM	RMI
MTS/COM+	EJB
ADO	JDBC
ADSI	JNDI
MSMQ	JMS
DTC	JTA/JTS
…	…

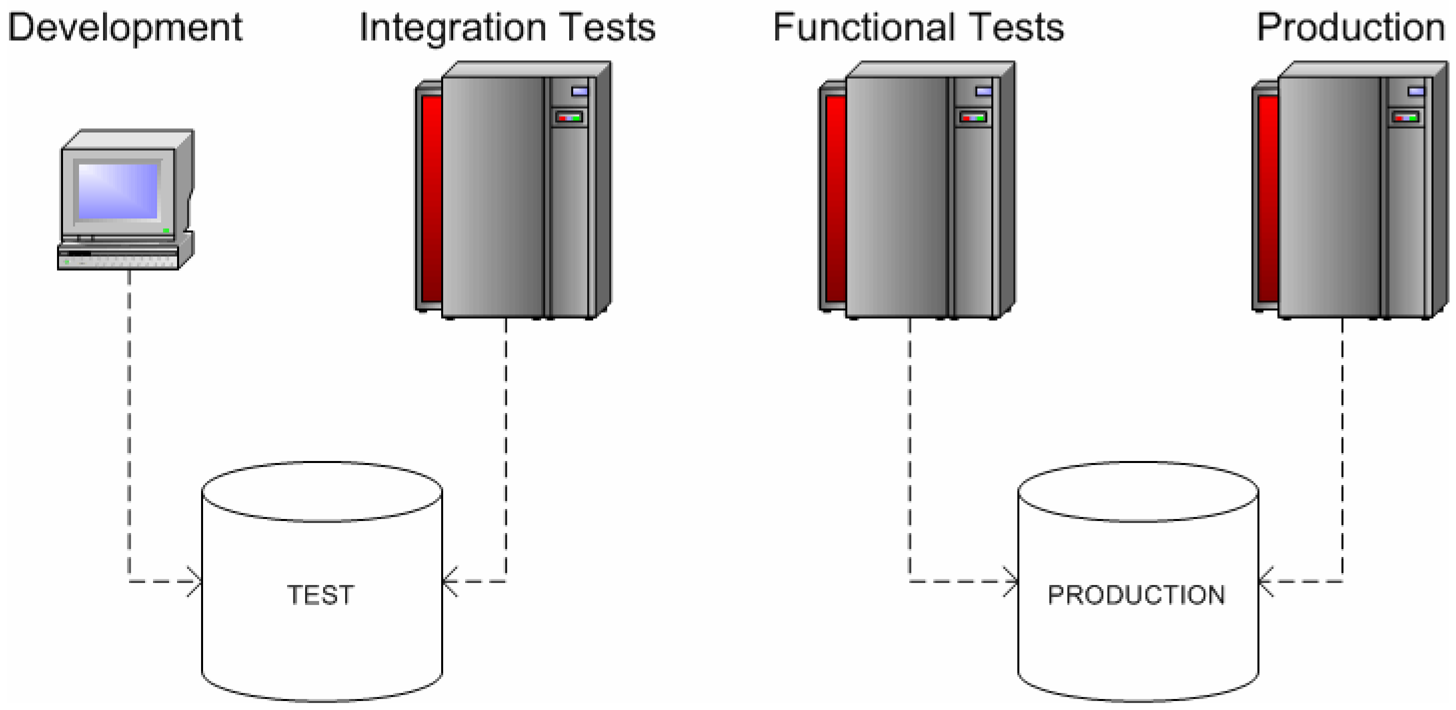
Common Patterns

Naming Services

- Map human-friendly names to objects
  - DNS
  - File System
  - LDAP

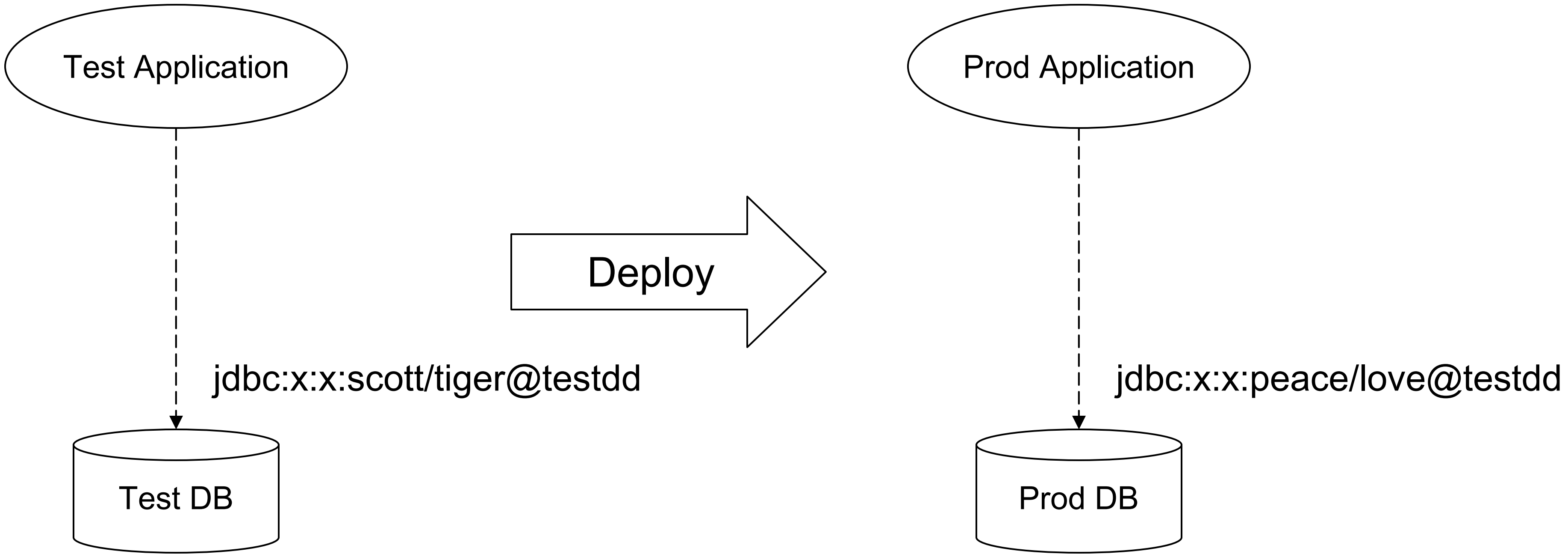
Adding this indirection layer we gain flexibility and portability.

Development and Deployment



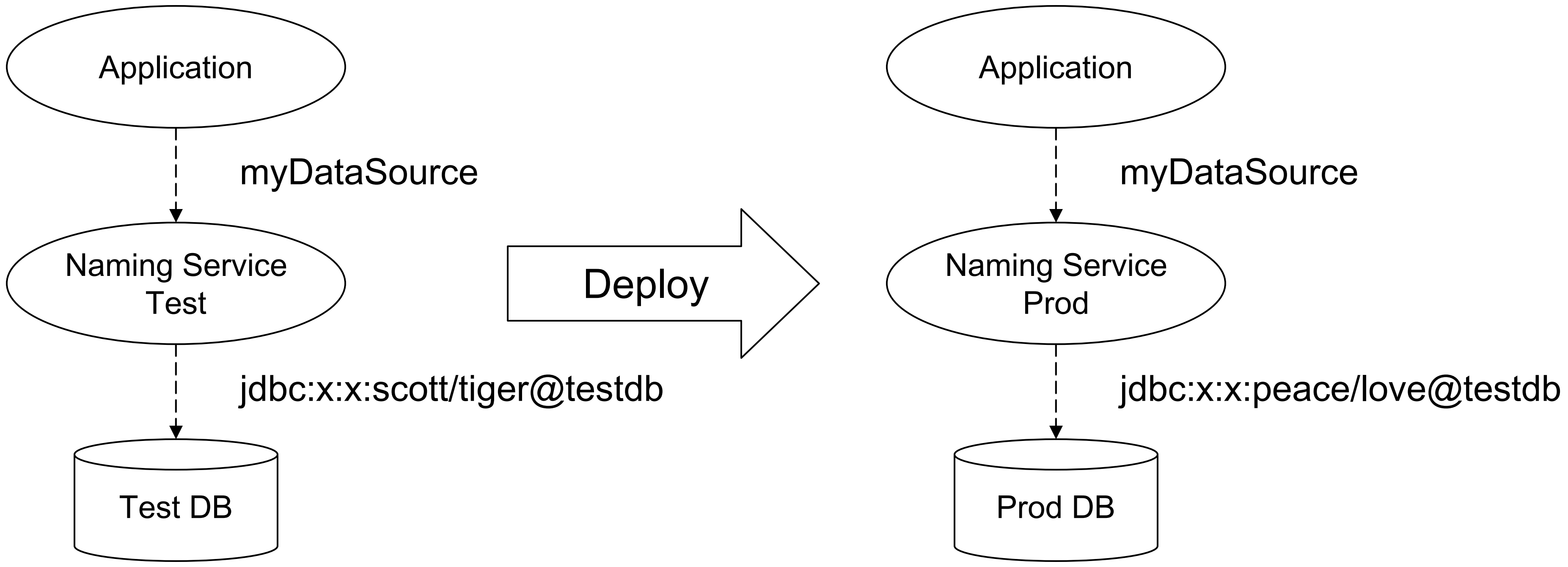
- Different Databases
- Different Hardware
- Different Operative Systems

Deployment dilemma



- There is a direct dependency between the application and the DB
- We must produce different “executables” for Test and Production environments
- Any change in the DB configuration will break our application

Enterprise Deployment



- No dependency between Application and DataBase
- No need for different Application versions
- Easier to maintain
- Separation of roles: Developer vs Application Server Administrator

Java Naming: JNDI

Java Naming and Directory Interface

Direct Connection

```
Class.forName("oracle.jdbc.driver.OracleDriver");
Connection conn =
DriverManager.getConnection("jdbc:x:x:scott/tiger@testdd");
/* use the connection */
conn.close();
```

JNDI Connection

```
Context ctx = new InitialContext();
Object dsRef=ctx.lookup("java:comp/env/jdbc/mydatasource");
DataSource ds=(Datasource) dsRef;
Connection conn=ds.getConnection();
/* use the connection */
conn.close();
```

JNDI Configuration

using JBoss

```
<datasources>
    <local-tx-datasource>
          <jndi-name>comp/env/jdbc/mydatasource</jndi-name>
          <connection-url>jdbc:x:x:@testdd</connection-url>
          <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
          <user-name>scott</user-name>
          <password>tiger</password>
    </local-tx-datasource>
</datasources>
```

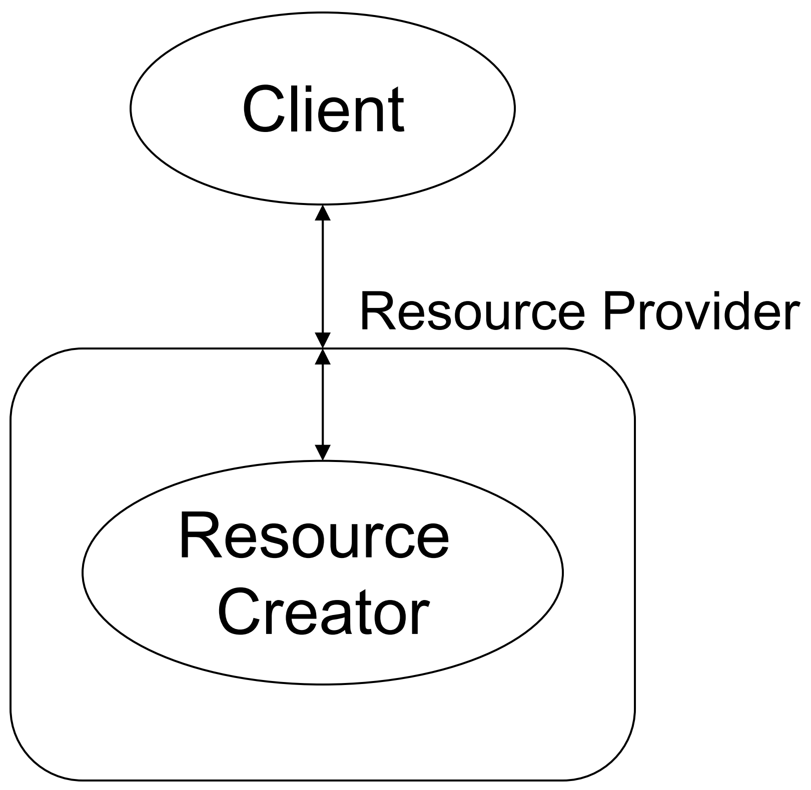
- Application Server administrator manages this
- Application Server specific

Pooling

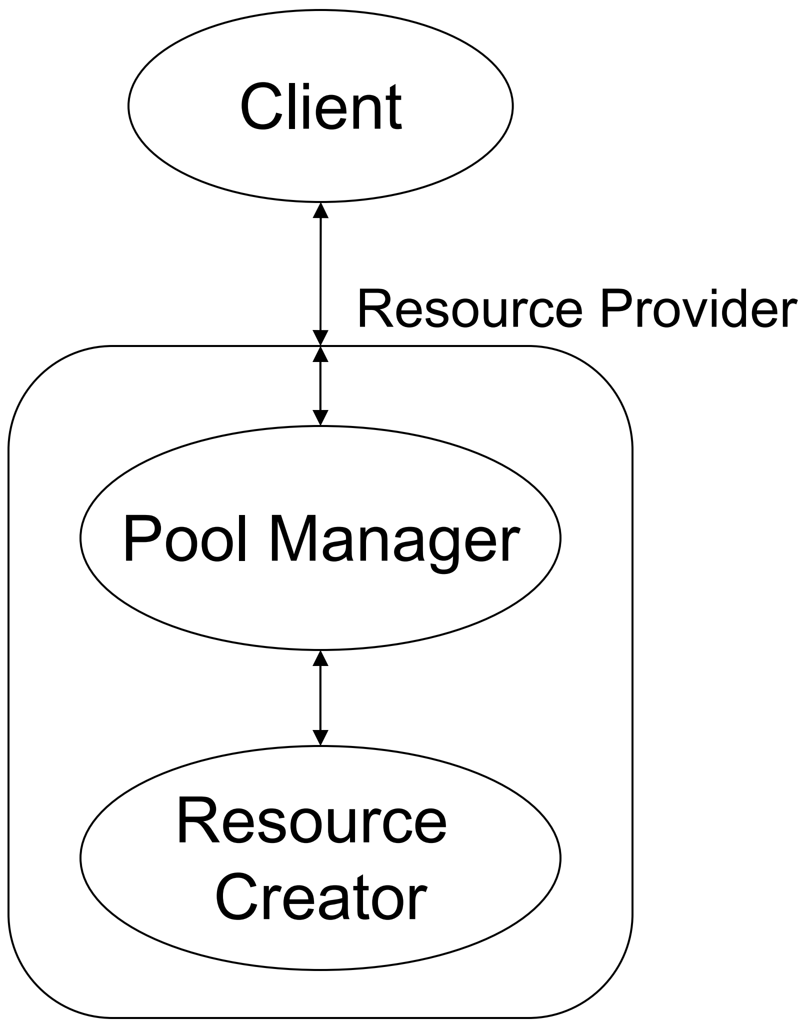
- *Pooling* means creating a pool of reusable resources
- Greatly improves performance if *creating* the resource is expensive (compared to *using* it)
- Should be completely *transparent* to the client
- Resources should be *stateless*

Pooling Schema

Without Pooling

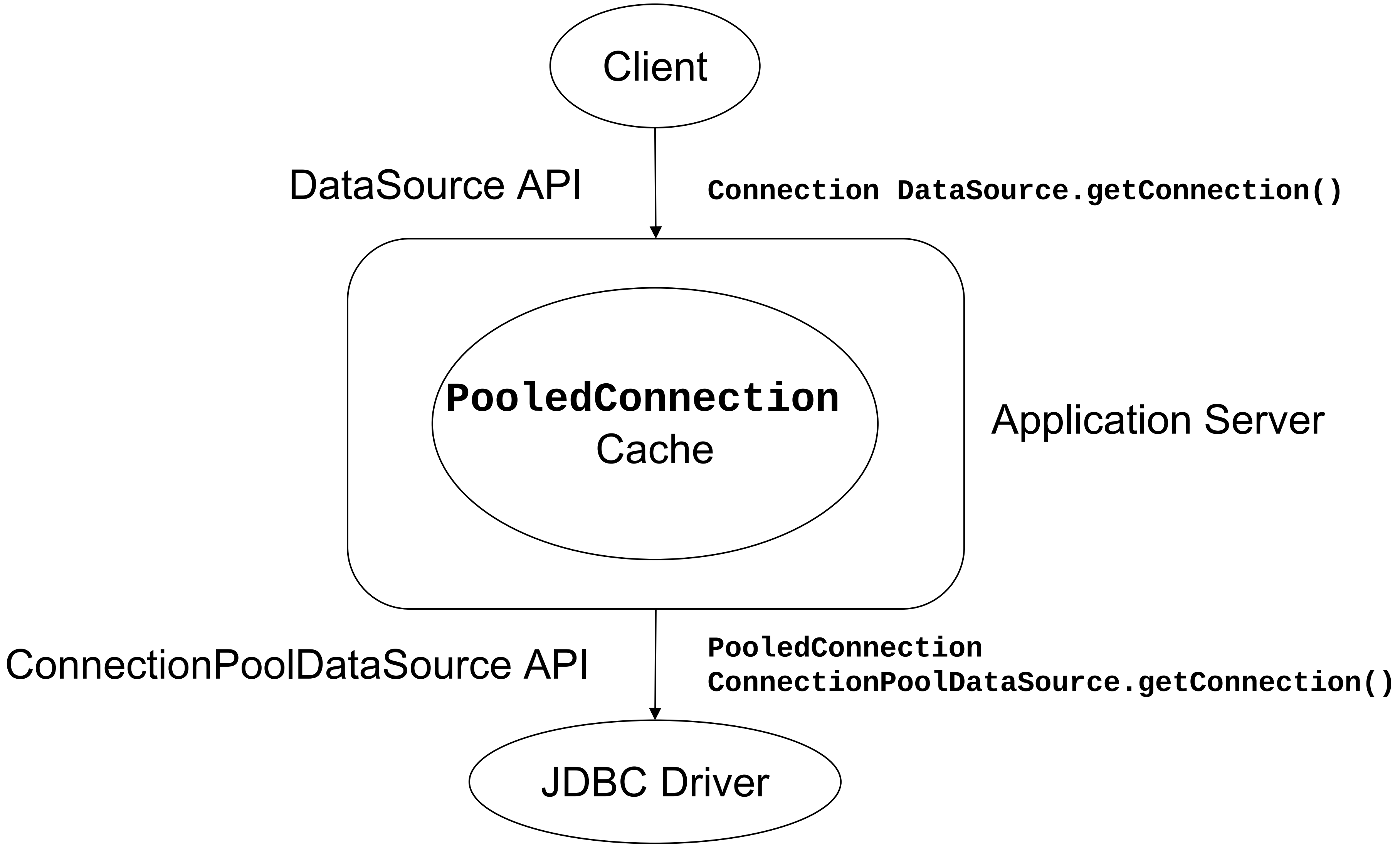


With Pooling



Java Pooling (JDBC)

Java DataBase Connectivity


Client
DataSource API
Connection DataSource.getConnection()
PooledConnection
Cache
Application Server
ConnectionPoolDataSource API
PooledConnection
ConnectionPoolDataSource.getConnection()
JDBC Driver

Pooling sequence

- Client requests new Connection
- DataSet obtains valid PooledConnection
  - From pool if any available
  - Otherwise create a new one
- Create new Connection from PooledConnection and return it to client
- When client closes the Connection the PooledConnection is *not* close but marked as available and returned to the pool.

Java Code Example

JNDI Connection + Pooling

```
Context ctx = new InitialContext();
Object dsRef=ctx.lookup("java:comp/env/jdbc/mydatasource");
DataSource ds=(Datasource) dsRef;
Connection conn=ds.getConnection();
/* use the connection */
conn.close();
```

- Same code as before!
- Complexity completely hidden to developer
- No need to change java sources when pooling parameters change

Pooling Configuration

with JBoss

```
<datasources>
    <local-tx-datasource>
        <jndi-name>comp/env/jdbc/mydatasource</jndi-name>
        <connection-url>jdbc:x:x:@testdd</connection-url>
        <driver-class>oracle.jdbc.driver.OracleDriver</driver-class>
        <user-name>scott</user-name>
        <password>tiger</password>

        <!-- Pooling parameters -->
        <min-pool-size>5</min-pool-size>
        <max-pool-size>100</max-pool-size>
        <blocking-timeout-millis>5000</blocking-timeout-millis>
        <idle-timeout-minutes>15</idle-timeout-minutes>
    </local-tx-datasource>
</datasources>
```

Transaction Management

What is a transaction?

An atomic unit of work. The work in a transaction must be completed as a whole; if any part of the transaction fails, the entire transaction fails.

Very well know problem that has been "solved" in databases for a long time.

ACID properties

Atomic: the transaction must behave as a single unit of operation. No partial work to commit

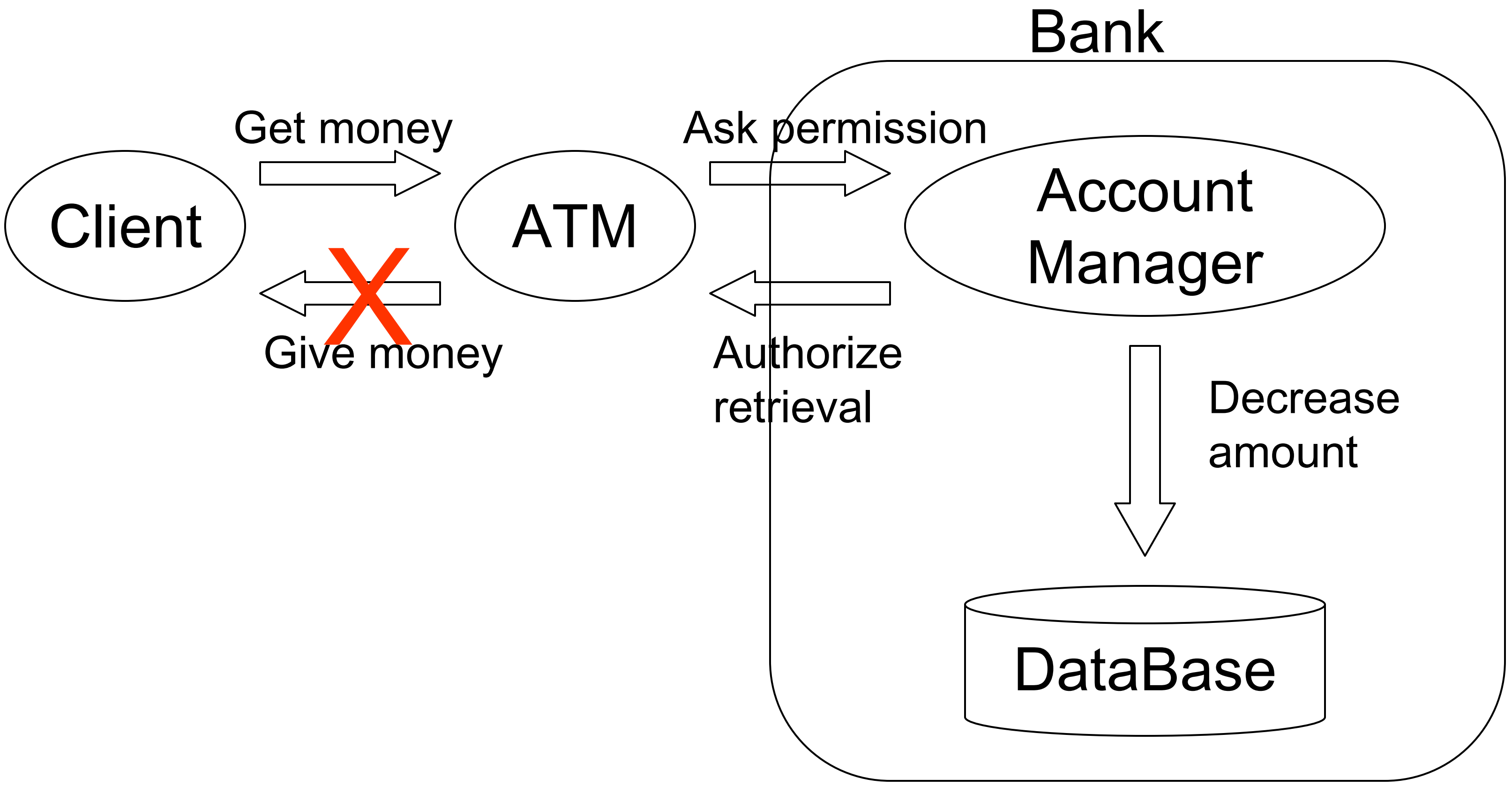
Consistent: either creates a new valid state or rolls back to the previous one

Isolated: a transaction in process and not yet committed must not interfere from all other concurrent transactions

Durable: committed data is saved in a way that the state can be restored even in case of system failure

SO/IEC 10026-1:1992 Section 4

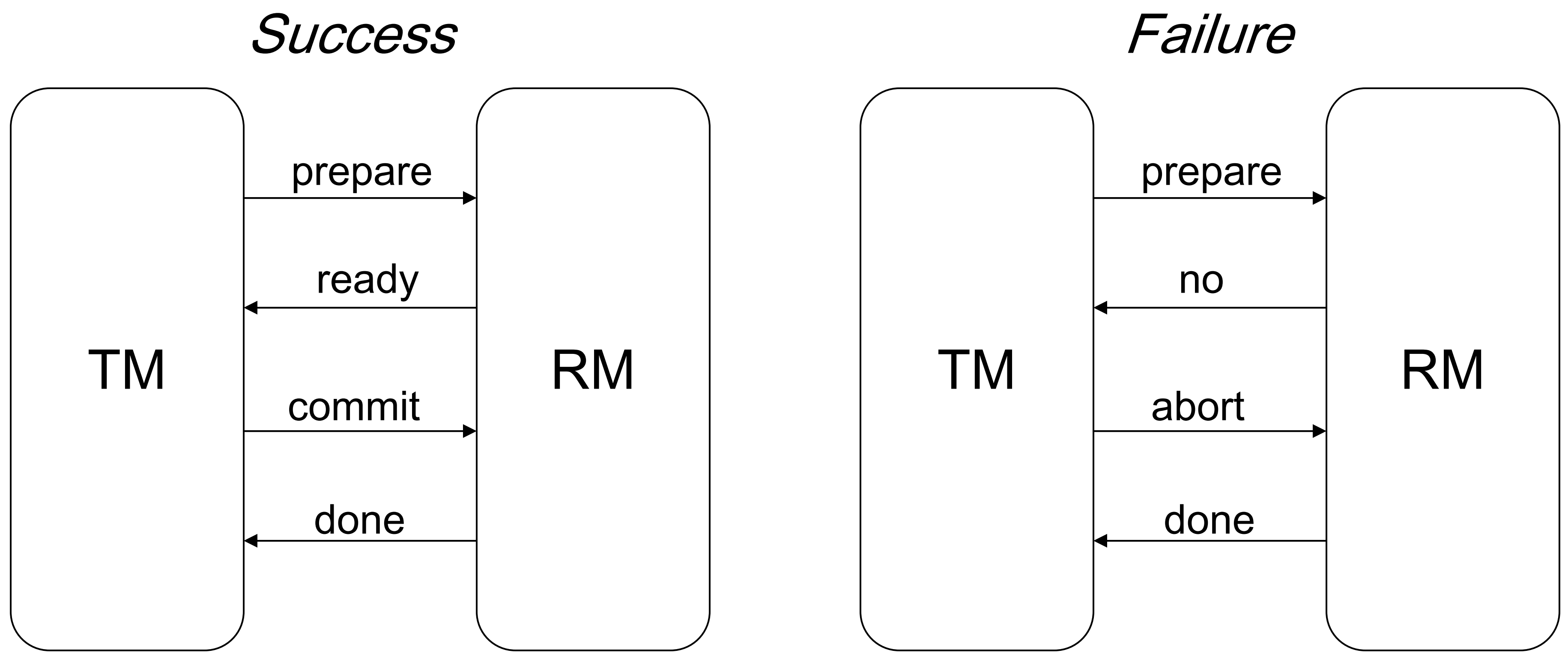
ATM Transaction example



We need to be able to manage distributed transaction to solve this class of problems.

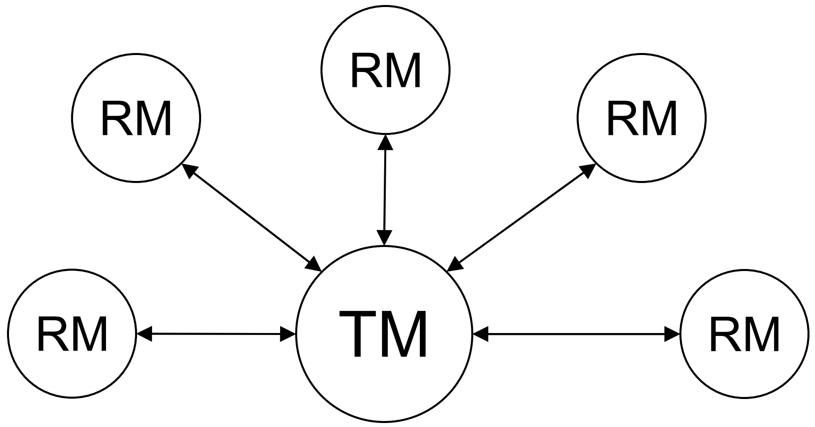
2-phase commit

- Transaction Manager [TM]
- Resource Manager [RM]



A log is kept for all operations, to let the TM recover a valid state in case of system failure

Distributed 2-phase commit



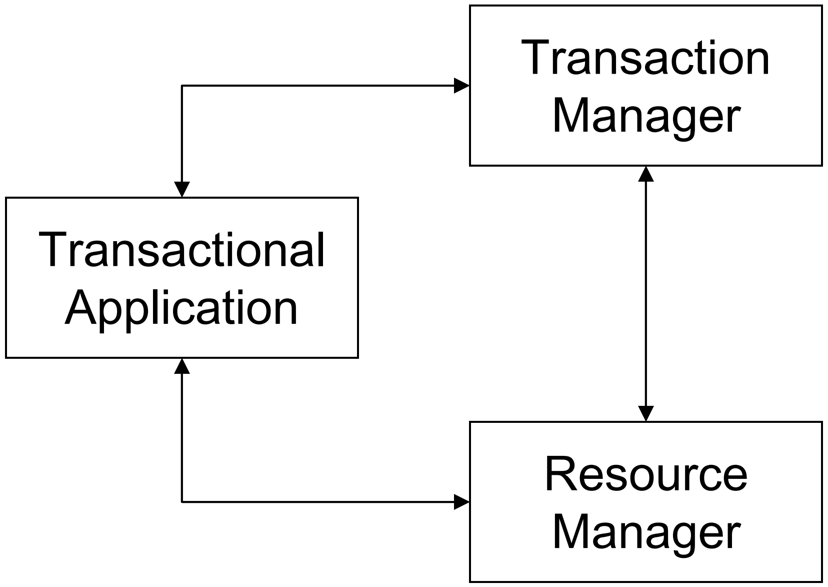
The TM repeats the 2-phase commit with every RM

- If the all RM answer "ready" the TM issues a global "commit"
- If at least one RM answers "no" the TM issues a global "abort"

Java Transactions (JTA)

Java Transaction API

Manage transactions in a *programmatic* way: you are responsible for programming transaction logic into your application code, that is calling begin(), commit(), abort().



```
Context ic = new InitialContext();
UserTransaction ut = (UserTransaction) ic.lookup(strTransJndi);
ut.begin();
// access resources transactionally here
ut.commit();
```

J2EE Declarative Transactions

It's possible to specify at *deploy time* the transaction behavior.

The Application Server will *intercept* calls to the components and automatically begin/end the transaction on your behalf

```
<ejb-jar>
    <enterprise-beans>
          <session>
                <ejb-name>SomeName</ejb-name>
                …
                <transaction-type>Container</transaction type>
          </session>
    </enterprise-beans>
</ejb-jar>
```

Transaction types

```
<container-transaction>
    <method>
        <ejb-name>myComponent</ejb-name>
        <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
</container-transaction>
```

The J2EE application server manages different managed transaction types:

- Required: always run in a transaction. Join the existing one or starts a new one
- RequiresNew: always starts a new transaction
- Supports: joins the client transaction if any. Otherwise runs in no transaction
- Mandatory: transaction must already be running. Otherwise throws exception
- NotSupported: doesn’t use transactions. Suspends client transaction if it exists
- Never: cannot be involved in a transaction. Throw exception if client has one

Conclusions

- There is no magic solution. You need experience to find the best compromise to solve your problem.
- You can solve any programming problem with an extra level of indirection ☺
- Except the problem of too many levels of indirection
- There are frameworks that already solve the most common and complex problems
- Understand the solution. Use the framework.
- Don’t reinvent the wheel

CERN School of Computing

Questions?

Resources

- J2EE tutorial (http://java.sun.com/j2ee/1.4/docs/tutorial/doc/)
- JBoss Docs (http://docs.jboss.org/jbossas/jboss4guide/r2/html/)
- Designing J2EE Apps
  (http://java.sun.com/blueprints/guidelines/designing_enterprise_applications_2e/DEA2eTOC.html)