Distributed Systems
Lecture 2: RPC-based Communication

Prof. Dr. Alexander Schill
http://www.rn.inf.tu-dresden.de
Outline

- RPC – Remote Procedure Call
  - Definition and Principle
  - Binding
  - Error processing

- RPC-Extensions
  - Asynchronous Calls
  - Mass Data Transfer
  - Callbacks
  - Local Calls

- RMI - Remote Method Invocation
Remote Procedure Call (RPC)

- Extension of procedure call to remote call
- Request/Response interaction scheme

- Goal: syntactic and semantic uniformity
  - Call mechanism
  - Language elements
  - Error semantics

- Definition (by Nelson)
  - Synchronous transfer of control thread
  - Level of programming language
  - Separate address spaces
  - Coupling via relatively narrow channels
  - Data exchange: call parameters and results
RPC: System Architecture

Example system: DCE (Distributed Computing Environment)

Distributed Systems – Lecture 2: RPC-based Communication
RPC: Interface Definition Language (IDL)

- Interfaces used by compiler to generate client and server stubs
- Stubs encapsulate functionality to:
  - transmit local call to remote computer
  - receive call on remote computer and pass on
  - convert data
    - before transmission on client (marshalling)
    - on arrival at remote computer (unmarshalling)

Example interface:

```idl
[uuid(765c3b10-100a-135d-1568-040034e67831),
 version(1.0),
]
interface ProductCatalogue  // Interface for product catalogue
{
  import "globaldef.idl";  // Import general definition
  const long maxDoc=10;   // Maximum number of products
  typedef [string] char *String;  // Data type for character strings
```
typedef struct {
  String productName;  // Product name
  String productType;   // Product type
  String productDescription;   // Textual Description
  Long size;           // Memory size
} productDescription; // Data structure ProductDescription

typedef struct {
  ProductDescription descr;    // Product description
  String header;               // Meta-Information
  char *data;                  // Product data
} Product;                    // Data structure Product

[idempotent] long searchProduct (   // Search products by type
  [in] String productType, // Input parameter: product type
  [out] ProductDescription *b[maxDoc],   // Output parameter: descriptions
  [out] long *status); // Execution status

long provideProduct (   // Provide a product
  [in] ProductDescription * descr, // Input parameter: description
  [out] Product *p); // Output parameter: product
RPC: Stub-Generation

Productcatalogue.idl

IDL-Compiler

Productcatalogue_cstub.o
Productcatalogue.h
Productcatalogue_sstub.o

#include

Productcatalogue_client.c
Productcatalogue_server.c

C compile

Productcatalogue_client.o
Productcatalogue_server.o

link

Productcatalogue_client.exe
Productcatalogue_server.exe

Distributed Systems – Lecture 2: RPC-based Communication
RPC: Binding process

Alternatives:
- Direct addressing
- Broadcast-request
- Directory Services

- **Client**
  - Import ProductCatalogue
  - Server Address S
  - Bind (S, ProductCatalogue)
  - RPC: provideProduct(productDesc)

- **Directory-Server**
  - Control Table

- **Server**
  - Export ProductCatalogue
    - searchProduct
    - provideProduct
    - addProduct
    - updateProduct
    - deleteProduct
  - Acknowledge (BindingNumber)
  - return(Product);
Caching of Binding Information
- information on the client-site is global for all processes
- recognition of out-dated information (for instance Timeout)
- limited binding information on the server-site (scalability, recovery/warm restart)

Time point of Binding
- compile time
- initialization time
- dynamic
- mixed techniques
  - logical names
  - first localization for binding during initialization time
  - re-localization due to errors

Flexibility vs. Effort
- **Standard transport protocols**
  - TCP/IP or UDP/IP
    - TCP already provides error processing, sequence ordering and duplicate recognition

- **Special transport protocols**
  - no connection setup (only implicit) \(\Rightarrow\) response time
  - active/passive connection state
  - only implicit connection release (timeout)
  - sharing of connections across different processes (only connection per machine \(\Rightarrow\) less connections)
Error semantics – describes error handling properties of an RPC system:

- **Maybe**
  - no error handling for lost messages
    - only for “non-important” operations

- **At least once**
  - at least once execution (if no machine crashes happen), lost messages repeated
  - only for idempotent operations, repetition after timeouts leads to duplicates

- **At most once**
  - duplicate recognition and removing; masks communication failures
  - one execution if no machine crashes happen

- **Exactly once**
  - exactly once execution
  - masks machine crashes, too
    - transaction concepts with restart and recovery
RPC: Error processing

(“?” means undetermined behaviour, depending on timing of possible failures)

<table>
<thead>
<tr>
<th>Error type</th>
<th>Error-free execution</th>
<th>Messages losses</th>
<th>Additional server failure</th>
<th>Additional client failure</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Maybe</strong></td>
<td>Execution: 1</td>
<td>Execution: ?</td>
<td>Execution: ?</td>
<td>Execution: ?</td>
</tr>
<tr>
<td><strong>At-Least-Once</strong></td>
<td>Execution: 1</td>
<td>Execution: &gt;=1</td>
<td>Execution: ?</td>
<td>Execution: ?</td>
</tr>
<tr>
<td></td>
<td>Result: 1</td>
<td>Result: &gt;=1</td>
<td>Result: ?</td>
<td>Result: ?</td>
</tr>
<tr>
<td><strong>At-Most-Once</strong></td>
<td>Execution: 1</td>
<td>Execution: 1</td>
<td>Execution: ?</td>
<td>Execution: ?</td>
</tr>
<tr>
<td><strong>Only-Once-Type-1</strong></td>
<td>Result: 1</td>
<td>Result: 1</td>
<td>Result: ?</td>
<td>Result: ?</td>
</tr>
<tr>
<td><strong>Exactly-Once</strong></td>
<td>Execution: 1</td>
<td>Execution: 1</td>
<td>Execution: 1</td>
<td>Execution: 1</td>
</tr>
<tr>
<td><strong>Only-Once-Type-2</strong></td>
<td>Result: 1</td>
<td>Result: 1</td>
<td>Result: 1</td>
<td>Result: 1</td>
</tr>
</tbody>
</table>
Problems of RPC

- Exchange of Client-/Server-roles
  - Rigorous client/server semantics
    - no equal communication partners
  - no interim result acknowledgments
  - no „callbacks“
- Multicast / Broadcast not supported at network level
- Strictly synchronous
- Standard RPC not appropriate for mass data transfer
- Transparency violations
  - variable parameters and data types (e.g. printf(“%s%d”,x1,x2);)
  - pointer parameters (e.g. char *x, ...)
  - global variables
  - error semantics
Thread usage for RPC

- Process control
  - Assignment of processes to procedure execution, deadlock handling
  - "lightweight" processes (Threads):
    - common address space
    - fast creation and process switching
    - large number of processes possible
      ⇒ use in RPC-Server-implementations
      ⇒ use in Client, too ⇒ asynchronous
  - Process assignment
    - process creation per call or
    - process-pool
  - Buffer transfer by reference via further protocol layers ⇒ efficiency
**Client-side:** Simultaneous calls on several servers

**Server-side:** Processing of several calls

**Example:**

Calls are processed parallel.
Asynchronous calls deliver a so called Future-Object to the Client
Acknowledgements (results) from the Server are delivered transparently to the respective Future-Object
Test- and receive operations on Futures enable the access of the Client to results of asynchronous RPCs
Special properties:
• full typing of Futures
• immediate sending of asynchronous call ⇒ response time optimization
RPC Extensions: Asynchronous RPCs

**Futuresets**

Client

- Futureset
  - Futures
    - ExtractReady
    - FClaim_

FInvoke_fetchOrder

return result

- Server 1
- Server 2
- Server 3
Futures in Java

- Future represents result of asynchronous task
- Implemented in `java.util.concurrent`
- `java.util.concurrent.Callable<V>`
  - task similar to Runnable but returns result and throws exception if unable to compute result

```java
public class MyCallableLong implements Callable<Long> {
    private type p1;
    private type p2;
    ...
    public void setParameters( p1, p2, ...) {...}

    public Long call() throws Exception {
        long result;
        // invoke a remote call which return a result of type long
        return result;
    }
}
```
Futures in Java

- Execution of Callable by thread provided by ExecutorService

ExecutorService executor = Executors.newFixedThreadPool(10);
Future<Long> future; //create future
Callable<Long> worker = new MyCallableLong(); //create worker
future = executor.submit(new MyCallable()); //execute Callable in thread

- Get result based on future

long result;
result = future.get(); //waits for the computation to complete
result = future.get(timeout, timeunit); //waits for at most the given time
for the computation to complete
future.isCanceled() //true, if this task was cancelled before it
completed normally
future.isDone() //true, if task completed (normal, canceled, exception)
future.cancel(boolean mayInterruptIfRunning) //cancel execution of task
RPC Extensions: Asynchronous RPCs

Time Behaviour

- RPC execution time: $t_s + t_a$
- $t_s$: roundtrip transfer time
- $t_a$: local server execution duration

Assumption

- Single machines for client and server with only one CPU/core
- Client and server can use multiple threads

Basic rule:

- $k$ asynchronous calls (with result) for the same server are at most
  \[
  \min \left( 1 + \frac{t_s}{t_a}, k \right)
  \]
  times faster than $k$ synchronous calls.

- $t_s \ll t_a \Rightarrow 1 + \frac{t_s}{t_a} \rightarrow 1 \Rightarrow$ no improvement
- $t_s \gg t_a \Rightarrow 1 + \frac{t_s}{t_a}$ gets large
  (for instance: for slower WANs)

- Maximum improvement $k$ times
  - calls are executed sequentially at the server site
  - $1x$ message roundtrip duration per call, if $t_a$ is small;
  - therefore max factor $k$ for $k$ calls (if $1 + \frac{t_s}{t_a}$ roundtrip gets larger than $k$)
RPC Extensions: Asynchronous RPCs

Time Behavior
- RPC execution time: \( t_s + t_a \)
- \( t_s \): roundtrip transfer time
- \( t_a \): local server execution duration

Assumption
- Single machines for client and server with only one CPU/core
- Client and server can use multiple threads

Basic rule:
- \( k \) asynchronous calls (with result) for the same server are at most
  \[
  \min (1 + \frac{t_s}{t_a}, k)
  \]
  times faster than \( k \) synchronous calls.

- \( t_s \ll t_a \Rightarrow 1 + \frac{t_s}{t_a} \rightarrow 1 \Rightarrow \) no improvement
- \( t_s \gg t_a \Rightarrow 1 + \frac{t_s}{t_a} \text{ gets large} \)
  (for instance: for slower WANs)

- maximum improvement \( k \) times
  - calls are executed sequentially at the server site
  - 1x message roundtrip duration per call, if \( t_a \) is small;
  - therefore max factor \( k \) for \( k \) calls (if \( 1 + \frac{t_s}{t_a} \) roundtrip gets larger than \( k \))
## RPC Extensions: Mass Data Transfer

<table>
<thead>
<tr>
<th>Optimization of response time</th>
<th>RPC</th>
<th>Protocols for mass data transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td>very important</td>
<td>lower importance</td>
<td></td>
</tr>
</tbody>
</table>

| Flow-control                 | lower importance | very important |

| Limitation of server-load   | very important   | lower importance |

| Storage of state information | should be avoided | of great importance for optimization goals |

| Protocol for connection setup | nonexistent or implicit | important for negotiating connection parameters |

| Initiation of data transfer  | immediately after initiation of a RPC | only after assembling large data volumes |
Synchronously embedded asynchronous calls (Casts)

- Casts do not have acknowledgments
- Selective repeat of casts after synchronization: at-most-once
- Explicit direction control via control token

Asynchronous data exchange

Client → Stub → Asynchronous data exchange → Stub → Server
Java Remote Method Invocation (RMI)

- RPC mechanism applied to object oriented model in Java
- All communication instances are uniformly represented as objects (local or remote)
- Reference and value parameter semantics supported; dynamic loading of class information for Remote Objects
- Calls are generally synchronous; asynchronous calls are possible via Threads
- Simple Naming Service: RMI Registry
  - Registration only local
  - Lookup local and remote
Java RMI: Example

Client

Object for User-Interaction

1. Invocation

2. Response with reference parameter to P

3. Loading of P class information

4. Invocation of P

5. Response

Server

Server-Object Product

C

Server-Object Product

P

I

1. Invocation

2. Response with reference parameter to P

3. Loading of P class information

4. Invocation of P

5. Response
- Generation of client and server stubs based on interface description
- Interfaces:
  - defined using the Java language
  - derived from the java.rmi.remote interface
  - inherit remote communication functionality

```java
public interface ProductCatalogue extends java.rmi.Remote {
    ProductDescription[] searchProduct(String productType)
        throws java.rmi.RemoteException;
    Product provideProduct(ProductDescription d)
        throws java.rmi.RemoteException;
    int deleteProduct(ProductDescription d)
        throws java.rmi.RemoteException;
    int updateProduct(Product p) throws java.rmi.RemoteException;
    ... }
```
Server-side implementation of interface

```java
public class ProductCatalogueImpl extends java.rmi.server.UnicastRemoteObject implements ProductCatalogue {

    public ProductCatalogueImpl() throws java.rmi.RemoteException {
        super();
    }

    public ProductDescription[] searchProduct(String productType) throws java.rmi.RemoteException {
        ProductDescription[] desc = ProductCatalogue.getDescriptionByType(productType);
        return desc;
    }

    ......
}
```

- Complex objects have to be serializable:
  ProductDescription implements java.io.Serializable
Server Implementation

```java
public class ProductCatalogueServer {
public ProductCatalogueServer() {
try {
    ProductCatalogue c = new ProductCatalogueImpl();
    Naming.rebind("rmi://localhost:1099/ProductCatalogueService", c);
} catch (Exception e) {...}
public static void main(String args[]) {
    new ProductCatalogueServer();
}}
```

Client implementation

```java
public class ProductCatalogueClient {
public static void main(String[] args) {
try {
    ProductCatalogue c = (ProductCatalogue)Naming.lookup("rmi://hostname/ProductCatalogueService");
    System.out.println(c.searchProduct("book"));
} catch (Exception e) {...}
}}
```
- **Proxy object**
  - a local representation of the remote object with an indirect reference
  - contains all methods in the interface description, but no application logic
  - contains information about location of remote objects
  - forwards calls to remote objects

- **Hash table - Logical object identification scheme**
Remote Reference Layer:
- control of remote object references
- activation of persistent objects if required

Transport Layer:
- connection control (as a rule one connection between a pair of operating system processes)
- object reference = <endpoint (IP-Address, Port); object ID>

Multithreaded Servers:
- default-mechanism for call execution of different objects
- calls of the same client are executed sequentially