2. RPC-based Communication
Outline

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

- **RPC-Extensions**
  - Asynchronous Calls
  - Mass Data Transfer

- **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
Example system: DCE (Distributed Computing Environment)

Import ← ... → Export

RPC: System Architecture
 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:

```plaintext
[ 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

Productcatalogue_client.c
Productcatalogue_server.c

Productcatalogue_client.h
Productcatalogue_server.h

Productcatalogue_client.o
Productcatalogue_server.o

Productcatalogue_client.exe
Productcatalogue_server.exe

#include
C compile
link
link
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
 RPC: Run-time support - Communication

- **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) ⇒ response time
  - active/passive connection state
  - only implicit connection release (timeout)
  - sharing of connections across different processes
    - (only connection per machine ⇒ 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>Error classes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>At-Least-Once</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>At-Most-Once</td>
<td>Execution: 1</td>
<td>Execution: 1</td>
<td>Execution: ?</td>
<td>Execution: ?</td>
</tr>
<tr>
<td>Only-Once-Type-1</td>
<td>Result: 1</td>
<td>Result: 1</td>
<td>Result: ?</td>
<td>Result: ?</td>
</tr>
<tr>
<td>Exactly-Once</td>
<td>Execution: 1</td>
<td>Execution: 1</td>
<td>Execution: 1</td>
<td>Execution: 1</td>
</tr>
<tr>
<td>Only-Once-Type-2</td>
<td>Result: 1</td>
<td>Result: 1</td>
<td>Result: 1</td>
<td>Result: 1</td>
</tr>
</tbody>
</table>

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

- Strictly synchronous -> asynchronous extensions
- Standard RPC not appropriate for mass data transfer -> enhanced protocols
- Strict, fixed client-/server-roles -> peer objects

- Transparency violations -> objects by reference
  - pointer parameters (e.g. char *x, ...)
  - global variables
Asynchronous extensions: Thread usage for RPC

- Lightweight processes (Threads):
  - common address space
  - fast creation and process switching
  - large number of processes possible
    - use in RPC-Server-implementations ⇒ parallel execution
    - use in Client, too ⇒ asynchronous invocation

- 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

1 call is waiting

1 thread is idle

(Thread)

Client 1

Client 2

Client 3

Client 4
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 (“isDone”, “get”).

Special properties:
- full typing of Futures
- immediate sending of asynchronous call ⇒ response time optimization
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

```java
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
  \[
  \min (1 + \frac{t_s}{t_a}; k)
  \]
  times faster than $k$ synchronous calls.

- $t_s << t_a \Rightarrow 1 + \frac{t_s}{t_a} \rightarrow 1 \Rightarrow$ no improvement
- $t_s >> t_a \Rightarrow 1 + \frac{t_s}{t_a}$ 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 $k$)
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
- $t_s \gg t_a \Rightarrow 1 + \frac{t_s}{t_a}$ 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></td>
<td>LOWER IMPORTANCE</td>
</tr>
</tbody>
</table>

| Flow-control                  |     | VERY IMPORTANT                    |
| LOWER IMPORTANCE              |     |                                  |

| Limitation of server-load     |     | LOWER IMPORTANCE           |
| VERY IMPORTANT                |     |                                  |

| Storage of state information  |     | OF GREATEST IMPORTANCE FOR OPTIMIZATION GOALS |
| SHOULD BE AVOIDED             |     |                                  |

| Protocol for connection setup |     | IMPORTANT FOR NEGOTIATING CONNECTION PARAMETERS |
| NONEXISTENT OR IMPLICIT       |     |                                  |

| Initiation of data transfer   |     | ONLY AFTER ASSEMBLING LARGE DATA VOLUMES |
| IMMEDIATELY AFTER INITIATION OF A RPC | |                                  |
RPC Extensions: Mass Data Transfer

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

Client → Stub → Asynchronous data exchange → Stub → Server
Object Model: 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

1. Invocation
2. Response with reference parameter to P
3. Loading of P class information
4. Invocation of P
5. Response

Client

Object for User-Interaction

UI

Server

Server-Object Product-catalogue

PC

Server-Object Product

P
- 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) {...}
    }
}
```
RMI: Java – Remote Reference Parameters

- References implemented by 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 location of remote objects (<endpoint (IP-Address, Port); object ID)

- Remote invocation
  - Proxy forwards calls to remote objects with guaranteed ordering
  - Efficient mapping of globally unique object ID onto local storage via hash table
  - Multithreaded server-side execution with guaranteed ordering

![Diagram showing remote object references and invocation process]