Distributed Systems

RPC-based Communication

Tenshi Hara
tenshi.hara@tu-dresden.de
https://www.rn.inf.tu-dresden.de/hara
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
Example system: DCE (Distributed Computing Environment)

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:

```
[ 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

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: product type
    [out] ProductDescription *b[maxDoc], // Output: descriptions
    [out] long *status); // Execution status

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

...
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 – 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)

**Server**
- Export ProductCatalogue
  - searchProduct
  - provideProduct
  - addProduct
  - updateProduct
  - deleteProduct
- Acknowledge (BindingNumber)
- return(Product);

**Directory-Server**
- Control Table
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

RPC: Binding Details

Flexibility vs. Effort
RPC: Run-time support - Communication

- Standard transport protocols
  - TCP/IP
    (TCP provides error processing, sequence ordering and duplicate recognition)
  - UDP/IP
    (Programmer needs to take further precautions)

- 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 cases:
1. Errors during procedure processing (similar to local procedure call)
2. Transfer errors

Error reasons:
- Message losses
- Failure of a participating computer
  - server-site → endless wait of client → Timeout
  - client-site → further processing via server as ‘orphan’
- Inaccessibility of the target node or outdated procedure-export information
  → solution: dynamic binding
• Maybe
  o single execution without notification in the case of errors
    → only for ‘non-important’ operations

• At least once
  o at least once execution (if no machine crashes happen)
  o only for idempotent operations

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

• Exactly once
  o exactly once execution
  o masks machine crashes, too
    → transaction concepts with warm restart and recovery
RPC: Error semantics

(‘?’: undetermined behaviour; depends 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>Only-Once-Type-1</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>Only-Once-Type-2</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 → unequal 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
- 1 call is waiting
- 1 thread is idle

RPC: Use of threads – General view
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 calls → response time optimization
FutureSets

Client

Futureset

Futures

...ExtractReady

FClaim...

FInvoke_fetchOrder

return result

Server 1

Server 2

Server 3

RPC Extensions: Asynchronous RPCs
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 remote call; expect type long result
        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

```java
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 (nor, can, err)
future.cancel(boolean mayInterruptIfRunning) // cancel execution of task
```
Time Behaviour
- RPC execution time: $t_s + t_a$
- $t_R$: roundtrip transfer time
- $t_E$: local server execution duration

Basic rule:
- $k$ asynchronous calls (with result) for the same server are at most

$$\min(1 + \frac{t_R}{t_E}; k)$$

times faster than $k$ synchronous calls.

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

- 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 $1+t_R/t_E$ roundtrip gets larger than $k$)
- $t_R \ll t_E \Rightarrow \lim(1 + t_R/t_E) \rightarrow 1$
  \Rightarrow \text{i.e., no improvement}

- $t_R \gg t_E \Rightarrow \lim(1 + t_R/t_E) \rightarrow \infty$
  \Rightarrow \text{i.e., execution significantly faster than communication (e.g., for lower WANs)}
<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
Local RPC-Optimization

- Issue: Overhead for remote communication
- Efficient RPCs within one computer
- Example: Lightweight RPC

![Diagram showing communication between two processes](image-url)

- Parameters in the common storage area
- Call without process exchange
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

Server

Server-Object Product-catalogue

Server-Object Product

C

P

I
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 serialisable:
     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

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
- maintains transparency for application
- provides indirect referencing of remote objects
- 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 OS 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
### Summary: Comparison of System Models

<table>
<thead>
<tr>
<th></th>
<th>RPC-Model</th>
<th>RMI Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>processing model</strong></td>
<td>procedural model with restrictions</td>
<td>object communication</td>
</tr>
<tr>
<td><strong>data access type</strong></td>
<td>indirect data access via RPC-Server</td>
<td>direct access to objects</td>
</tr>
<tr>
<td><strong>data transfer</strong></td>
<td>value parameter semantics</td>
<td>reference parameter semantics</td>
</tr>
<tr>
<td><strong>identity</strong></td>
<td>not system-wide unique</td>
<td>system-wide unique</td>
</tr>
<tr>
<td><strong>granularity</strong></td>
<td>server of coarse granularity</td>
<td>objects of any granularity</td>
</tr>
<tr>
<td><strong>placement</strong></td>
<td>fixed placement</td>
<td>modifiable placement</td>
</tr>
</tbody>
</table>

**higher transparency grade and improved influence on distribution with object-oriented model**

