Lecture 12 · SENG 41283 · Distributed and Cloud Computing

Remote Invocation — Deep Dive

Message Identifiers · Failure Model · History · R/RR/RRA Protocols · HTTP Methods · RPC Implementation · RMI Design

1. Request–Reply Protocol

Designed to support roles and message exchanges in typical client–server interactions. Synchronous: the client blocks until the reply arrives. Reliable: the reply is effectively an acknowledgement. Built over UDP to avoid redundant acknowledgements, TCP connection overhead, and flow-control overhead.

1.1 Message Identifiers

Every message has a unique identifier consisting of two parts: (1) a requestId from an increasing sequence of integers unique to the sender (reset at $2^{32}-1$); (2) an identifier for the sender process (port and Internet address) — unique in the distributed system.

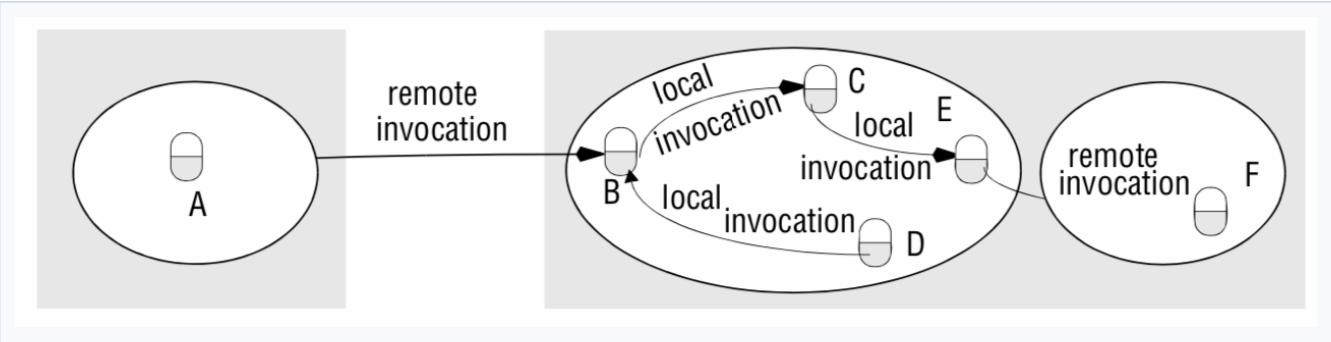

Figure 1.1 Message identifier structure: requestId (unique to sender) and sender process identifier (unique in the distributed system).

1.2 Failure Handling

- **Timeouts:** client retransmits request repeatedly until reply received or server assumed failed.
- **Duplicate filtering:** recognises messages from the same client with the same requestId and filters out duplicates.
- **Idempotent operations:** can be repeated with the same effect (e.g. add to set). Non-idempotent: append to sequence.
- **History:** structure storing (requestId, message, clientId) for retransmission without re-execution.

1.3 Exchange Protocols

Name	Messages sent by		
	Client	Server	Client
R	Request		
RR	Request	Reply	
RRA	Request	Reply	Acknowledge reply

Figure 1.2 Exchange protocol variants: R (request only), RR (request–reply), RRA (with acknowledgement).

• **R:** single Request; no reply; client proceeds immediately; used when no confirmation is needed.

• **RR:** server reply IS the acknowledgement; failures masked by retransmission + duplicate filtering + history.

• **RRA:** three messages; Acknowledge reply enables server to discard history entries.

2. HTTP Methods

Method	Description
GET	Retrieve resource at given URL; no side effects.
HEAD	Identical to GET but returns only status line and headers.
POST	Send data to server; used for form submission, mailing lists, and database append.
PUT	Store data at given URL; modification or new resource.
DELETE	Server deletes the resource at given URL.
OPTIONS	Server returns list of methods allowed for given URL.
TRACE	Server sends back the request message; used for diagnostics.

3. RPC Implementation

RPC enables calling a function on a remote server using the same syntax as a local function call. Benefits of programming with interfaces: abstraction from implementation details, language/platform independence, natural support for software evolution.

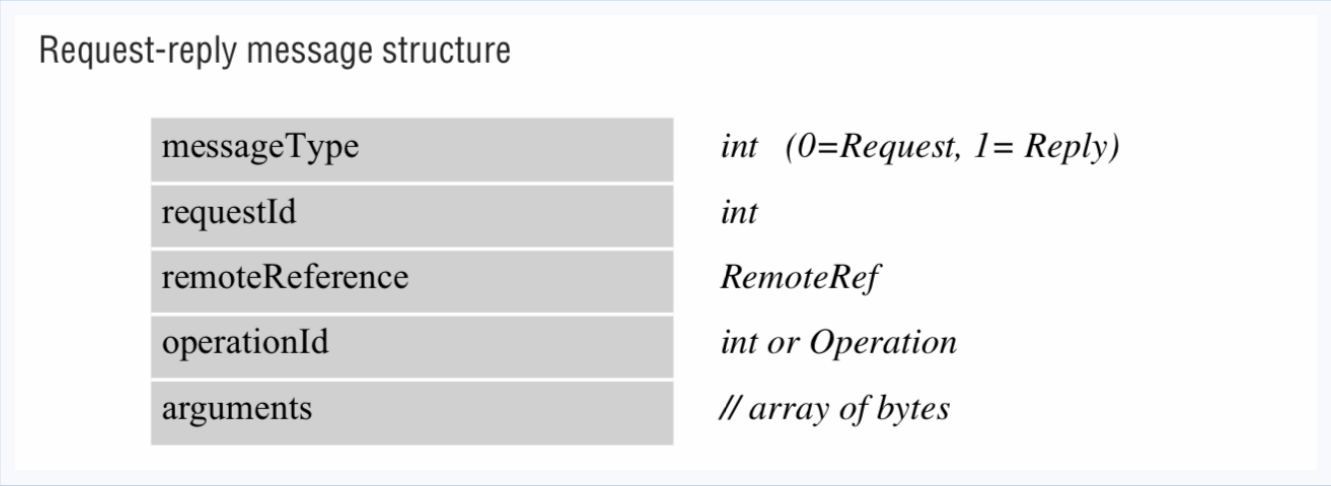

Figure 3.1 RPC implementation: client stub, dispatcher, server stub, and service procedures; stubs are auto-generated from the interface definition.

RPC Components

- **Client stub:** marshals procedure ID and arguments; sends; unmarshals results on reply.
- **Dispatcher:** selects server stub by procedure identifier.
- **Server stub:** unmarshals arguments; calls service procedure; marshals return values.
- Stubs and dispatcher are generated automatically by an interface compiler.

Python rpyc Example

```
# Start local RPC server: python bin/classic.py
import rpyc
conn = rpyc.classic.connect('localhost')
conn.execute("print('Hello from RPC')")
# Output: Hello from RPC
```

4. Remote Method Invocation (RMI)

RPC extended to distributed objects. Commonalities with RPC: both support programming with interfaces; both built on request–reply protocols with at-least-once and at-most-once semantics; similar transparency.

Additional Expressiveness

- Full OOP: objects, classes, inheritance, design methodologies.
- Unique object references for all objects (local or remote); references can be passed as parameters.

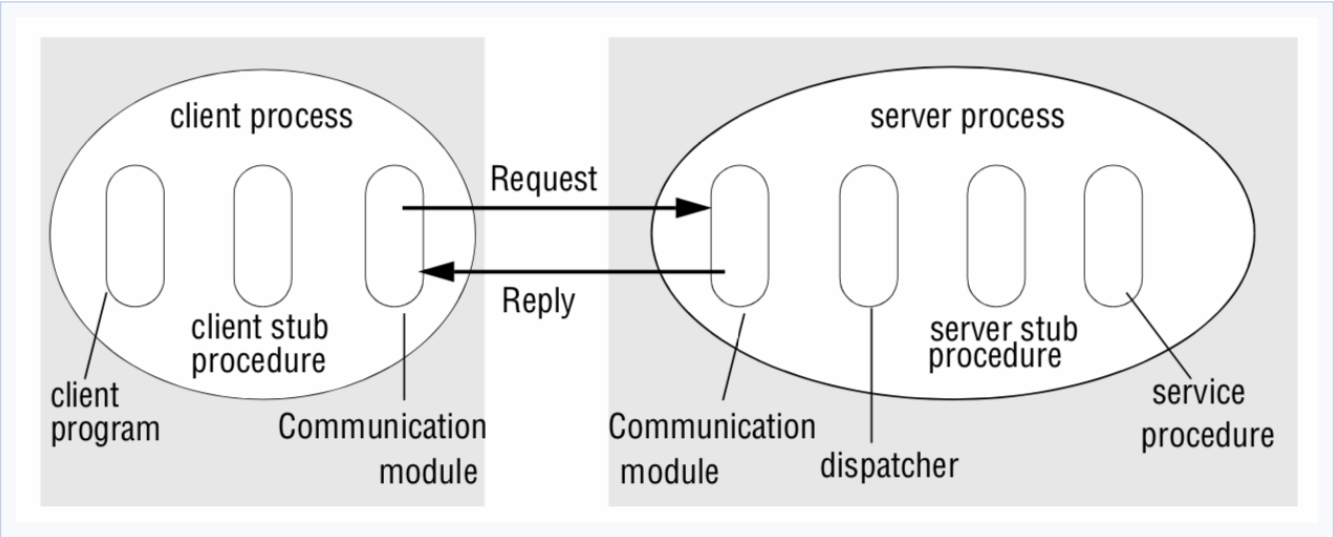

Figure 4.1 RMI design elements: object references, interfaces, actions, exceptions, and garbage collection.

The Distributed Object Model

Each process contains objects; some receive both local and remote invocations, others only local invocations. Method invocations between objects in different processes are remote method invocations.

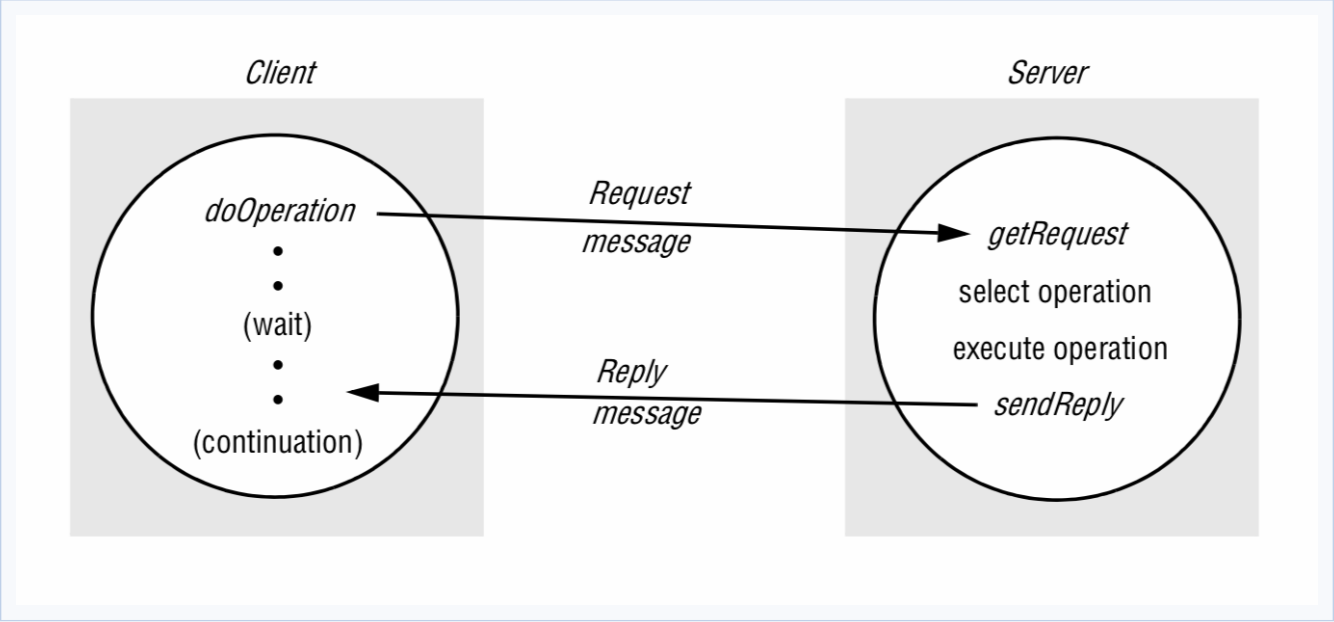

Figure 4.2 Distributed object model: objects A, B (remote interface), C, D, E, F (remote interface) across two processes.