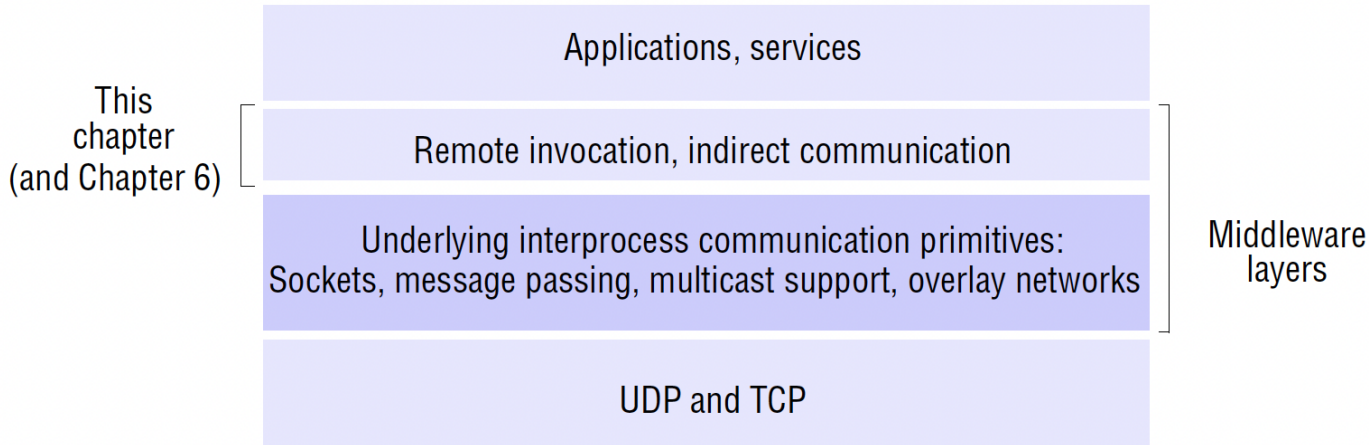
Cloud computing and distributed systems

Chapter #5
Remote invocation

Zeynep Yücel

Ca' Foscari University of Venice
zeynep.yucel@unive.it
yucelzeynep.github.io

Introduction

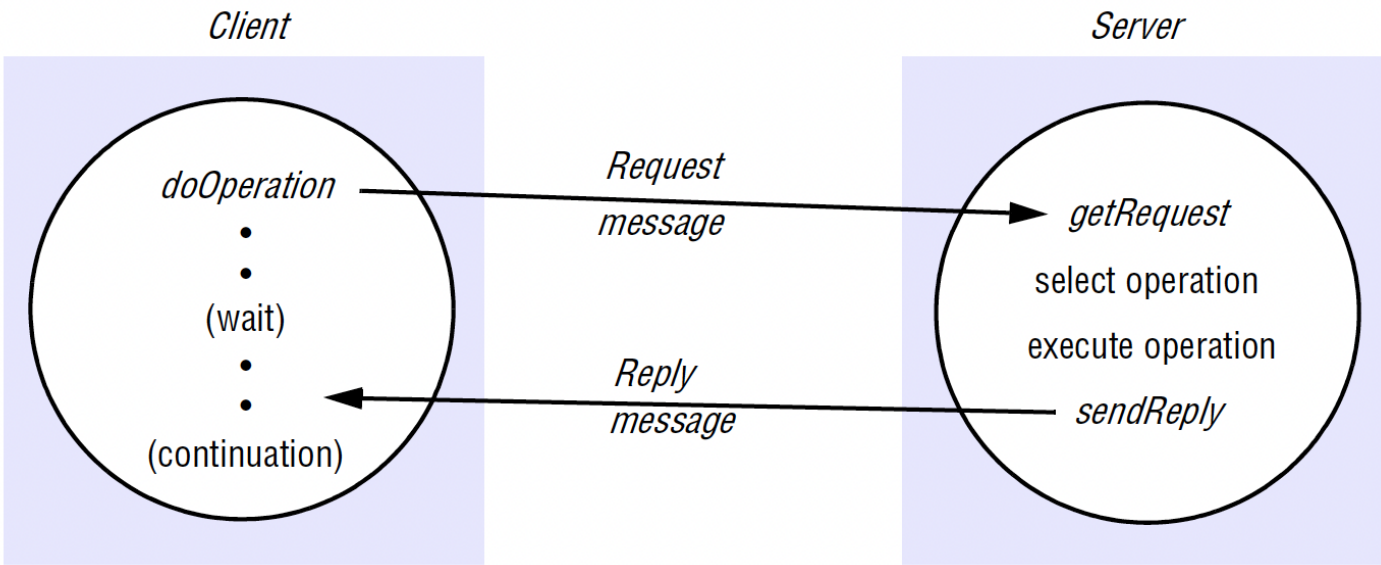


- Entity communication methods.
  - Request-reply protocols.
  - Remote Procedure Call.
  - Remote Method Invocation.
- Middleware and communication styles.

Request-reply protocols

- Synchronous or asynchronous.
  - Synchronous: blocking.
  - Asynchronous: non-blocking.
- Client-server via UDP datagrams.
  - Acknowledgements unnecessary.
  - Two additional messages for establishing communication.
  - Flow control unnecessary.

Request-reply protocols



- Three main methods: doOperation, getRequest, sendReply.
- Matches requests and replies, ensures delivery.
- Protocol ensures delivery, if UDP datagrams are used.
  - Server reply as acknowledgement.

Request-reply protocols

Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
Sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();
Acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
Sends the reply message *reply* to the client at its Internet address and port.

- doOperation: Invoke remote operations with specified arguments.
  - Client marshals arguments and unmarshals results.
  - RemoteRef specifies address and port.
  - Client sends the request and waits for reply (blocking).

Request-reply protocols

Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)
Sends a request message to the remote server and returns the reply.
The arguments specify the remote server, the operation to be invoked and the arguments of that operation.

public byte[] getRequest ();
Acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);
Sends the reply message *reply* to the client at its Internet address and port.

- getRequest: Acquire service requests.
- SendReply: after processing.
- Upon receipt, client resumes execution.

Request-reply protocols

Request-reply message structure

messageType	*int (0=Request, 1= Reply)*
requestId	*int*
remoteReference	*RemoteRef*
operationId	*int or Operation*
arguments	*// array of bytes*

- Message structure
  - Message type.
  - Unique requestId.
    - RequestId in reply.
    - Verify correspondence.
  - Remote reference.
  - Operation to be invoked.
    - Operations numbered.

Request-reply protocols

Message identifiers

- Identifiers for reliable delivery.
- Two parts of identifier:
  - RequestId.
  - Sender process identifier.
- Ensures uniqueness overall.

Request-reply protocols

Failure model of the request-reply protocol

- UDP communication failures.
  - Omission failures.
  - Unordered delivery.
- Processes can crash.
- Timeouts to handle server failures and lost messages.

Request-reply protocols

Timeout and Discarding

- After timeout, fail immediately or keep trying.
- Return immediately with failure indication.
  - Not common.
  - Lost messages.
- Resend request until reply or detecting lack of service.
  - Server filters duplicates.

Request-reply protocols

Lost reply messages

- Duplicate requests received for replies already sent.
- Unless original result is stored, execute operation again.
- Idempotent operations: one that can be performed repeatedly without changing the outcome.
  - E.g., set addition.
- If only idempotent messages, no measures necessary.

Request-reply protocols

History

- Stored results.
- History: Record of replies.
  - Request identifier, message, client identifier.
- Resend replies.
- Memory cost.
  - Store last reply to each client.
  - If many clients, storage problem.

Request-reply protocols

Styles of exchange protocols

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

- Exchange protocols.
  - Request (R) protocol
    - Single Request.
    - No response or confirmation needed.
    - Client proceeds immediately.
  - Request-Reply Protocol.
    - Involves two messages.
    - Reply acts as acknowledgment.
    - Failure handling: Retarnsmission, history.
  - Request-Reply-Acknowledge Protocol.
    - Involves three messages.
    - Acknowledge reply.
    - More resources used.

Request-reply protocols

Use of TCP streams to implement the request-reply protocol

- Buffer size decision.
- Datagram size limitations.
- TCP streams for variable size.
- Reliable delivery and flow control.
  - Java serialization.
  - No need to deal with retransmission, filtering duplicates etc.
  - Reduced overhead.
- Trade-off.

Request-reply protocols

HTTP: An example of a request-reply protocol

- HTTP
- Web servers manage resources.
- Client requests structure.

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

- HTTP defines message structure.
  - Content negotiation.
  - Authentication challenges.

Request-reply protocols

HTTP: An example of a request-reply protocol

- HTTP over TCP.
- Traditionally: Connection inefficiency.
- HTTP 1.1: Persistent connections.
- Request/reply in ASCII text strings.
- MIME types structure.

Request-reply protocols

HTTP methods

- Common HTTP methods.
  - GET: Retrieve data.
  - HEAD: Header only.
  - POST: Send data.
    - Submit forms.
    - Mailing list posts.
    - Database updates.
  - PUT: Store or modify.
  - DELETE: Remove resource.
  - OPTIONS: Allowed methods.
  - TRACE: Request diagnostics.
- PUT and DELETE idempotent, POST not idempotent.

Request-reply protocols

Message contents

method	*URL or pathname*	*HTTP version*	*headers*	*message body*
GET	http://www.dcs.qmul.ac.uk/index.html	HTTP/ 1.1		

- HTTP Request message.
  - Request headers.

HTTP version	*status code*	*reason*	*headers*	*message body*
HTTP/1.1	200	OK		resource data

- HTTP Reply message.
  - Status code in 3 digits.
  - E.g. redirection (303).
- Message body contains data.

RPC and design issues

- Simplifies programming.
- Similar to local calls.
- Hides complexities.
- Key concepts: interface programming, call semantics, transparency.

Remote Procedure Call (RPC)

Interfaces in distributed systems

- Modular programming.
- Modules on separate processes.
- Server procedures.
- Service interface.
- Interface benefits.
  - Abstraction.
  - Independence from language or platform.
  - Software evolution.

Remote Procedure Call (RPC)

Interfaces in distributed systems

- No direct access to variable in another process
- Local call mechanisms unsuitable.
- Addresses cannot be passed.
- Influences interface language design.

Remote Procedure Call (RPC)

Interface Definition Languages (IDLs)

- RPC mechanisms integrate with programming languages.
- If all parts written in the same language, convenience in local and remote invocation.
- Multiple language support beneficial.
- IDLs enable cross-language calls.
- Examples of IDLs listed.

Remote Procedure Call (RPC)

RPC call semantics

- Implementation variations for different delivery guarantees.
  - Retry message determines resend.
  - Duplicate filtering manages retransmissions.
  - Result retransmission controls lost results.

Remote Procedure Call (RPC)

RPC call semantics

Fault tolerance measures			*Call semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

- Local calls execute exactly once.
- RPC invocation semantics involve:
  - Maybe
  - At-least-once
  - At-most-once

Remote Procedure Call (RPC)

Reliability semantics - Maybe semantics

- Maybe semantics: once or not.
- No fault-tolerance measures apply.
  - Omission failures
  - Crash failures
- If request message is lost, procedure is not executed.
- If no result message arrives by timeout, it is uncertain if the procedure ran or not.
- Acceptable for occasional failures.

Remote Procedure Call (RPC)

Reliability semantics - At-least-once semantics

- Either result (executed at least once) or exception.
- Achieved by retransmission.
- Can suffer from:
  - Crash failures.
  - Arbitrary failures.
    - Non-idempotent operations.

Remote Procedure Call (RPC)

Reliability semantics - At-most-once semantics

- Either result (executed exactly once) or exception.
- All fault-tolerance measures.
- Retries manage failures.
- Sun RPC uses at-least-once.

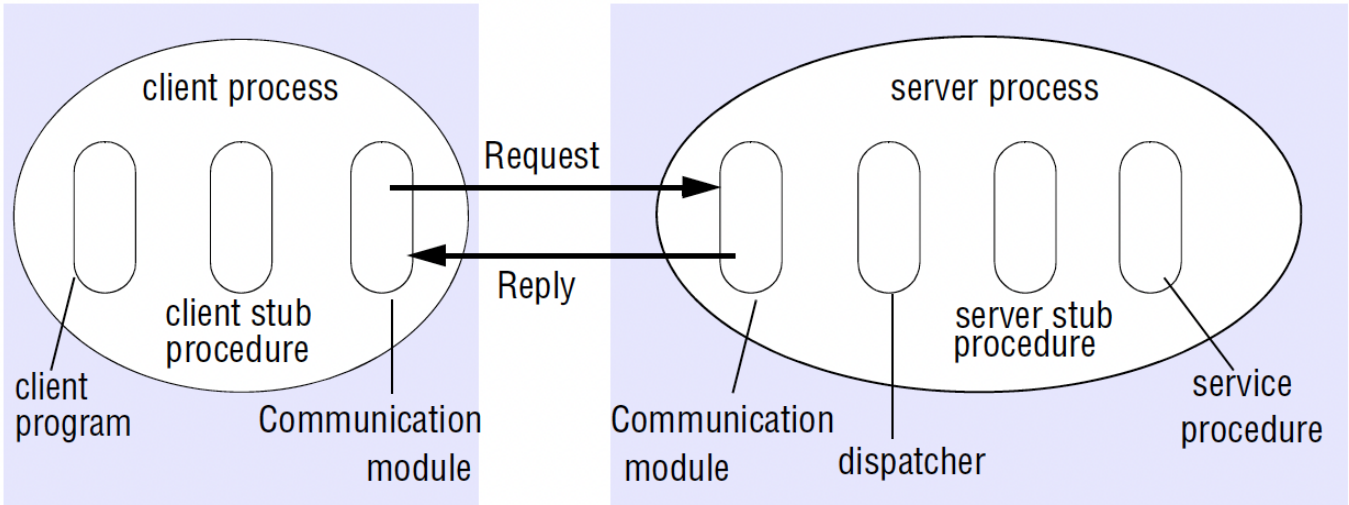
Remote Procedure Call (RPC)

Transparency

- Remote calls similar to local ones.
- Location and access transparency.
  - Hides physical location.
  - Access remains consistent.
- Remote calls more prone to failures than local calls.
- Higher latency than local.
- Mechanism necessary to abort calls that take too long.
  - Syntax consistency maintained, differences reflected to service interface.
  - Differences should be clear.
  - Different syntax for clarity.
  - IDLs can specify exception handling and call semantics.

Remote Procedure Call (RPC)

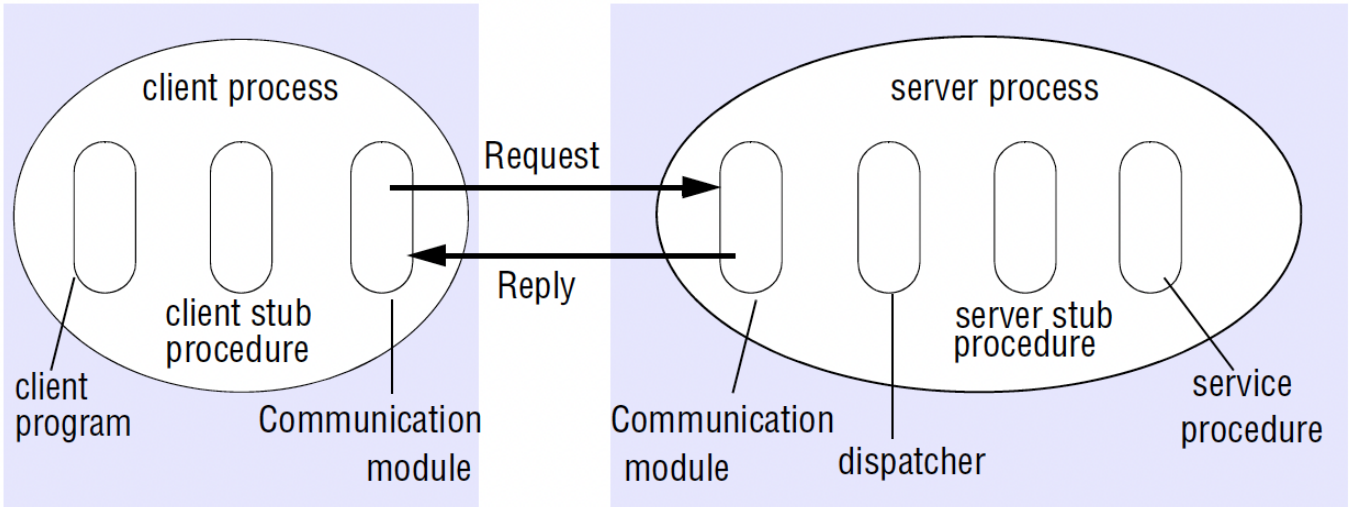
Implementation of RPC



- Uses request-reply protocol.
- Client utilizes stub procedure.
- Stub marshals procedure identifier and arguments and unmarshals the result message.

Remote Procedure Call (RPC)

Implementation of RPC



- Server contains a dispatcher, server stub procedure and service procedure.
- Dispatcher selects server stub.
- Server stub unmarshals arguments, calls service procedure and marshals the reply.
- Client/server stubs procedures and dispatcher can be generated from interface.

Remote method invocation (RMI)

- RMI extends RPC capabilities.
- Remote object method invocation.
- Similarities with RPC:
  - Interface support.
  - Request-reply protocols.
  - Similar level of transparency.
- Differences:
  - Full use expressive power object-oriented programming.
  - Unique references for objects.
  - Parameter passing by value or by reference.

Remote method invocation (RMI)

Design issues for RMI

- The object model:
  - Multiple interacting objects.
  - Objects communicate by invoking each others' methods (passing arguments, receiving results).
  - Direct access to instance variables.
  - Data accessible only through methods.

Remote method invocation (RMI)

Object References and Interfaces

- Object References:
  - Objects accessed via references.
  - Variables hold references.
  - To call a method, specify reference, method, arguments.
  - Target or receiver.
  - First-class values.
- Interfaces
  - Defines method signatures.
  - Provides specific interface.
  - Multiple interfaces can be implemented by a class.
  - No constructors for interfaces.

Remote method invocation (RMI)

Actions

- Actions: method calls across objects.
  - May include arguments.
  - Effects of invocation:
    - Change receiver's state.
    - Create new instantiated.
    - Further invocations.

Remote method invocation (RMI)

Exception and Garbage collection

- Exceptions: error management.
  - Indicate potential errors.
  - Control shifts on error.
- Garbage collection frees up memory (e.g. in Java).
  - Manual memory management in C++ etc.

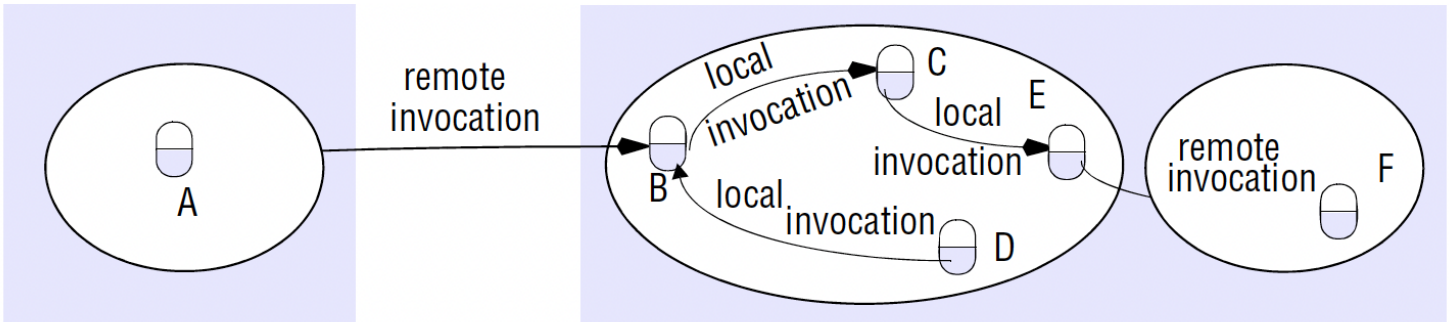
Remote method invocation (RMI)

Distributed objects

- Logical partitioning by objects.
- Client-server architecture.
  - Servers manage objects.
  - Clients invoke methods.
  - Servers can act as clients to allow chains of invocations.
- Advantages of architecture:
  - Replication for fault tolerance.
  - Encapsulation enforced.
  - Access via RMI or cache.

Remote method invocation (RMI)

Distributed object model

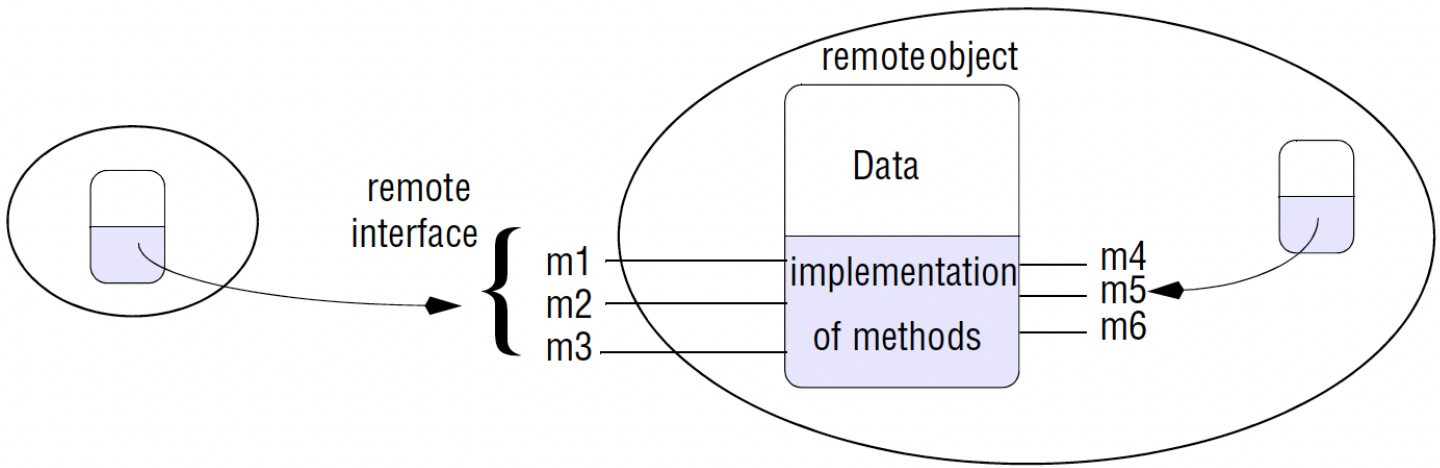


- Some objects with both local and remote calls, some objects with only local calls.
- Remote objects: one that can receive remote invocations.
- Remote invocations between objects in different processes.
- To invoke RMI, remote object reference is necessary.

Remote method invocation (RMI)

Remote object references and remote interfaces

- Unique identification of objects by remote object references.
- Local versus remote.
- Remote object reference as argument or result.

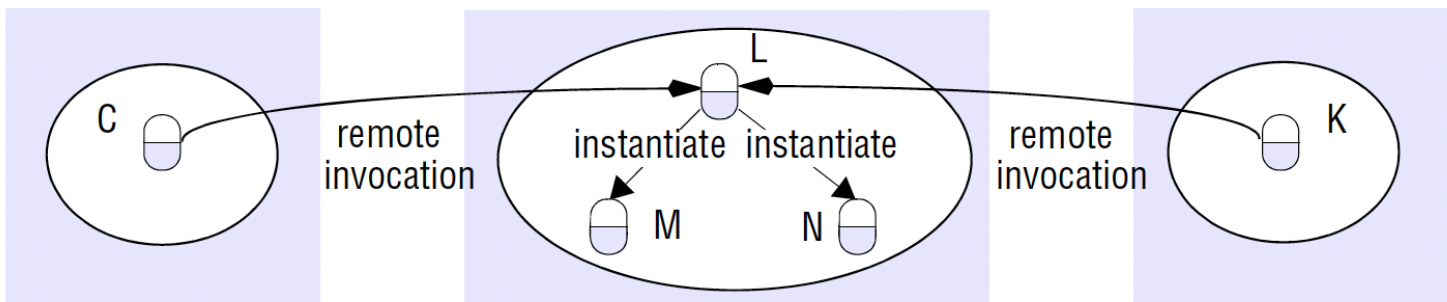


- Local objects can call remote interface methods or others.
- Objects in other processes can invoke only remote interface methods.

Remote method invocation (RMI)

Actions on distributed systems

- Actions start with method invocation.
- Invocations of methods in different processes or computers.
- RMI for crossing process or computer boundaries.
- Remote reference necessary for RMI.
- Remote object instantiation, e.g. invocations from C and K lead to instantiation of M and N.



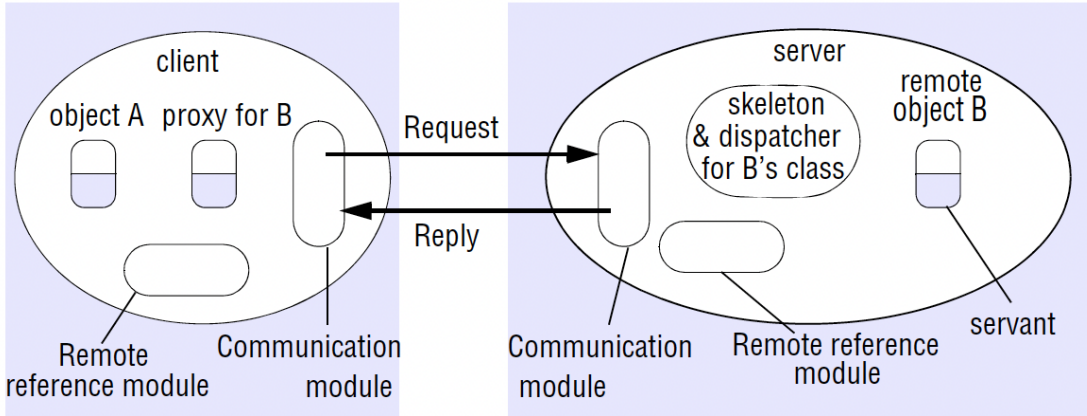
Remote method invocation (RMI)

Garbage collection and exception

- Garbage collection in a distributed object system.
  - Cooperation with the existing local garbage collector.
  - Manual deletion necessary, if garbage collector is not supported.
- Distribution exceptions.
  - Failures possible due to process crash, message loss etc.
  - Raise exceptions accordingly.

Remote method invocation (RMI)

Implementation of RMI - Communication module



- Object A calls.
- Communication modules used.
- Message types included.
- Dispatcher selects methods.

Remote method invocation (RMI)

Remote reference module

- Remote reference table
- Local and remote references
- At first use, create a new reference and add it to the table.
- Retrieve local references from remote object references.
- Used in un/marshalling.

Remote method invocation (RMI)

Servants and RMI software

- Servant handles remote requests.
  - Valid until not needed.
- Middleware function.
- Proxy hides details.
  - Forwards remote calls.
- Dispatcher selects methods.
- Skeleton implements methods.

Remote method invocation (RMI)

Generation of the classes for proxies, dispatchers and skeletons and Dynamic invocation

- Proxy, dispatcher, skeleton by interface compiler.
- Proxy not available in compile time?
- Dynamic invocation.
- Useful when interface cannot be predicted at compile time.

Remote method invocation (RMI)

Distributed garbage collection

- Object alive when it is referenced
- Memory reclaimed when object is unreferenced
- Java garbage collection:
  - Reference counting
  - Server tracks references
  - Client informs server before creating proxy
  - Client notifies removal when object is not necessary
  - Until holder list empty

Discussion topic

Explain the design choices that are relevant to minimizing the amount of reply data held at a server. Compare the storage requirements when the RR and RRA protocols are used.

Discussion topic

The Election interface provides two remote methods:

vote: with two parameters through which the client supplies the name of a candidate (a string) and the 'voter's number' (an integer used to ensure each user votes once only). The voter's numbers are allocated sparsely from the range of integers to make them hard to guess.

result: with two parameters through which the server supplies the client with the name of a candidate and the number of votes for that candidate.

The Election service must ensure that a vote is recorded whenever any user thinks they have cast a vote.

Discuss the effect of maybe call semantics on the Election service.

Would at-least-once call semantics be acceptable for the Election service or would you recommend at-most-once call semantics?

Discussion topic

A client makes remote procedure calls to a server. The client takes 5 milliseconds to compute the arguments for each request, and the server takes 10 milliseconds to process each request. The local OS processing time for each send or receive operation is 0.5 milliseconds, and the network time to transmit each request or reply message is 3 milliseconds. Marshalling or unmarshalling takes 0.5 milliseconds per message.
Calculate the time taken by the client to generate and return from two requests:
(i) if it is single-threaded,
(ii) if it has two threads that can make requests concurrently on a single processor.
You can ignore context-switching times and assume that the server is single- threaded.