



Communication in Distributed Systems

Part 1

Corso di Sistemi Distribuiti e Cloud Computing
A.A. 2021/22

Valeria Cardellini

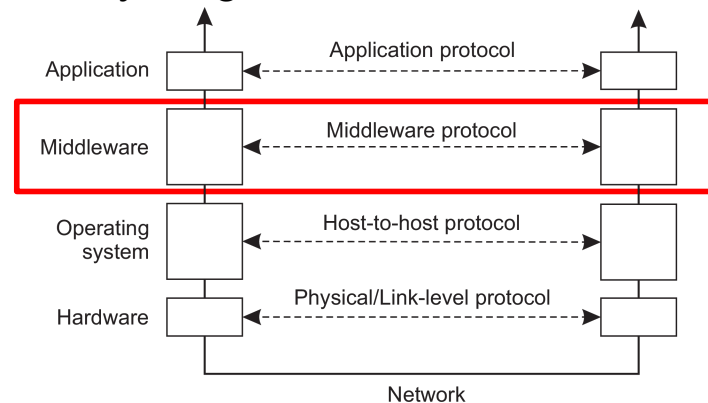
Laurea Magistrale in Ingegneria Informatica

Communication in distributed systems

- Based on **message passing**
 - Send and receive messages
- To allow for message passing, parties must agree on **many low-level details**
 - How many volts to signal a 0 bit and how many for a 1 bit?
 - How many bits for an integer?
 - How does the receiver know which is the last bit of the message?
 - How can the receiver find if a message has been corrupted and what to do then?

Basic networking model and its adaptation

- We know the solution: **divide network communication in layers**
 - The well known ISO/OSI reference model
 - We don't care about low-level details: for many distributed systems, the lowest-level interface is that of the network layer
- An adapted layering scheme



Middleware protocols

- **Middleware layer:** provides common services and general-purpose protocols
 - high-level
 - independent of specific applications
 - can be used by other applications
- Some examples:
 - **Communication protocols:** to call remote procedures or remote methods, queue messages, support multicasting and streaming
 - **Naming protocols:** to share resources among applications
 - **Security protocols:** to allow applications to communicate securely
 - **Distributed consensus protocols,** including distributed commit (to establish that a transaction is completed by all involved parties or has no effect)
 - **Distributed locking protocols**
 - **Data consistency protocols**

Types of communication

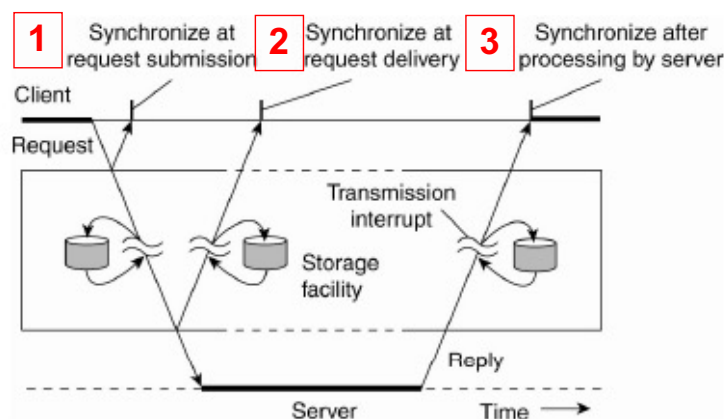
- Let's distinguish
 - **Persistency**
 - Transient versus persistent communication
 - **Synchronization**
 - Synchronous vs. asynchronous communication
 - **Time dependence**
 - Discrete vs. streaming communication

Persistent vs. transient communication

- **Persistent** communication
 - Message is stored by communication middleware as long as it takes to deliver it to receiver
 - Sender does not need to continue execution after submitting the message
 - Receiver does not need to be executing when message is submitted
- **Transient** communication
 - Message is stored by middleware only as long as sender and receiver are executing: sender and receiver have to be active at time of communication
 - If delivery is not possible, message is discarded
 - Transport-layer example: routers store and forward, but discard if forward is not possible

Synchronous communication

- Once the message has been submitted, sender is **blocked** until operation is completed
- Send and receive are **blocking** operations
- How long is sender blocked? Alternatives:
 1. until middleware takes over request transmission
 2. until message is delivered to receiver
 3. until message is fully processed by receiver



Valeria Cardellini - SDCC 2021/22

6

Asynchronous communication

- Once message has been submitted, sender continues its processing: message is temporarily stored by middleware until it is transmitted
- Send is a non-blocking operation, receive can be blocking or non-blocking

Valeria Cardellini - SDCC 2021/22

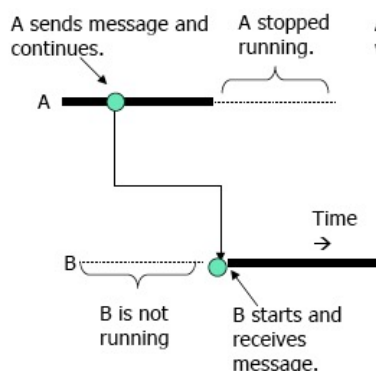
7

Discrete vs. streaming communication

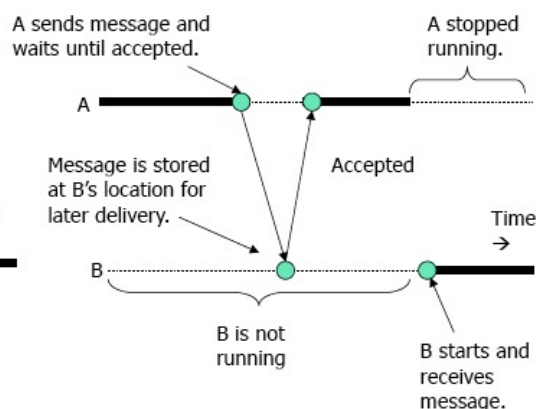
- Discrete communication
 - Each message forms a complete unit of information
- Streaming communication
 - Involves sending multiple messages, in temporal relationship or related to each other by sending order, which is needed to reconstruct complete information

Combining communication types

- Combination of **persistence** and **synchronization**
 - a) **Persistent and asynchronous** communication
 - E.g., email, Teams chat
 - b) **Persistent and synchronous** communication
 - Sender blocked until the (guaranteed) message copy is delivered to receiver



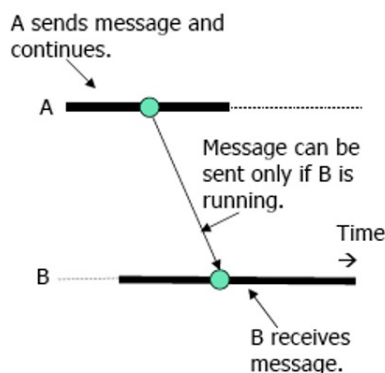
(a) Persistent asynchronous communication



(b) Persistent synchronous communication

Combining communication types

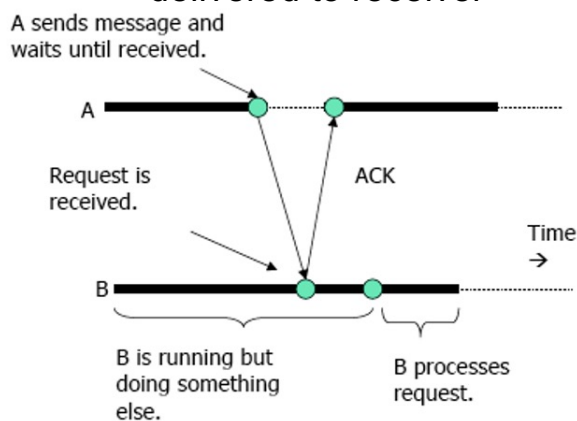
- Combination of **persistence and synchronization**
- c) **Transient and asynchronous** communication
 - Sender does not wait but message can be lost if receiver is unreachable (e.g., UDP)



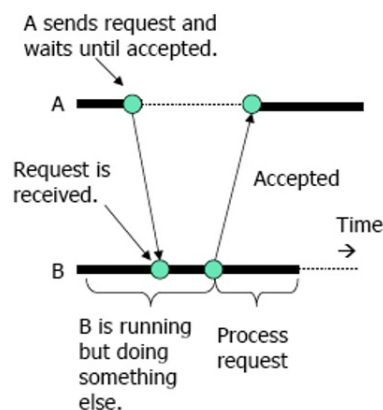
(c) Asynchronous communication

Combining communication types

- Multiple options for **transient and synchronous** communication
- d) **Receipt-based synchronous**: sender blocked until message copy is in receiver space (e.g., asynchronous RPC)
- e) **Delivery-based synchronous**: sender blocked until message is delivered to receiver



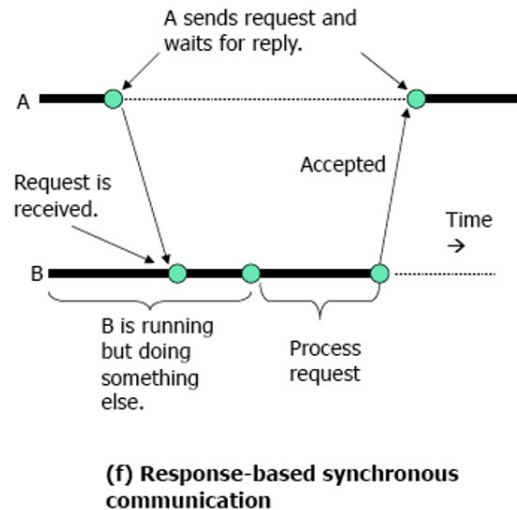
(d) Receipt-based synchronous communication



(e) Delivery-based synchronous communication

Combining communication types

- Multiple options for **transient and synchronous** communication
 - f) **Response-based synchronous**: sender blocked until it receives a reply message from receiver (e.g., synchronous RPC and RMI)



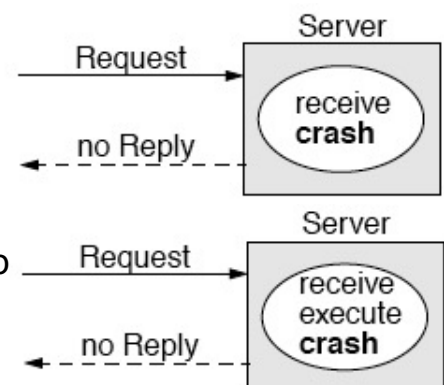
Failure semantics during communication

- Different types of failure in communication between sender (client) and receiver (server)
 - Request and/or reply message can be lost or delayed, or connection reset

- *Network is reliable* ("The Eight Fallacies of Distributed Computing")

2. Server can crash

- before performing the requested service
 - after performing the requested service
- client cannot distinguish between a and b



3. Client can crash

Failure semantics during communication

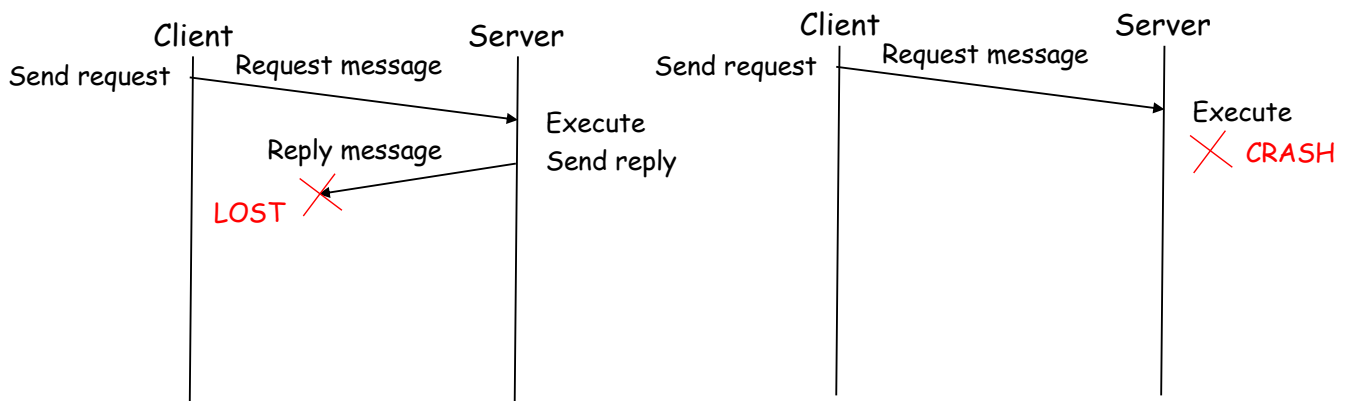
- What is the semantics of communication in the presence of failures in a DS?
 - **May-be** semantics
 - **At-least-once** semantics
 - **At-most-once** semantics
 - **Exactly-once** semantics
- Failure semantics applies both to service processing (e.g., RPC) and message delivery (e.g., MOM)
 - Let's focus on service processing

Basic mechanisms for failure semantics

- Failure semantics depends on the combination of the following 3 basic mechanisms
 1. Client side: **Request Retry (RR1)**
 - Client keeps trying until it gets a reply or is confident about server failure after a certain number of failed retries
 2. Server side: **Duplicate Filtering (DF)**
 - Server discards any duplicate request from the same client
 3. Server side: **Result Retransmit (RR2)**
 - Server keeps result to be able to retransmit it without recalculating in case it receives a duplicate request
 - Needed if operation performed by server is **not idempotent**
- Idempotent operation** (i.e., without side effects): multiple executions of the operation produce the same effect/result as a single execution of the operation, e.g., read-only operation or $x = 1$

May-be semantics

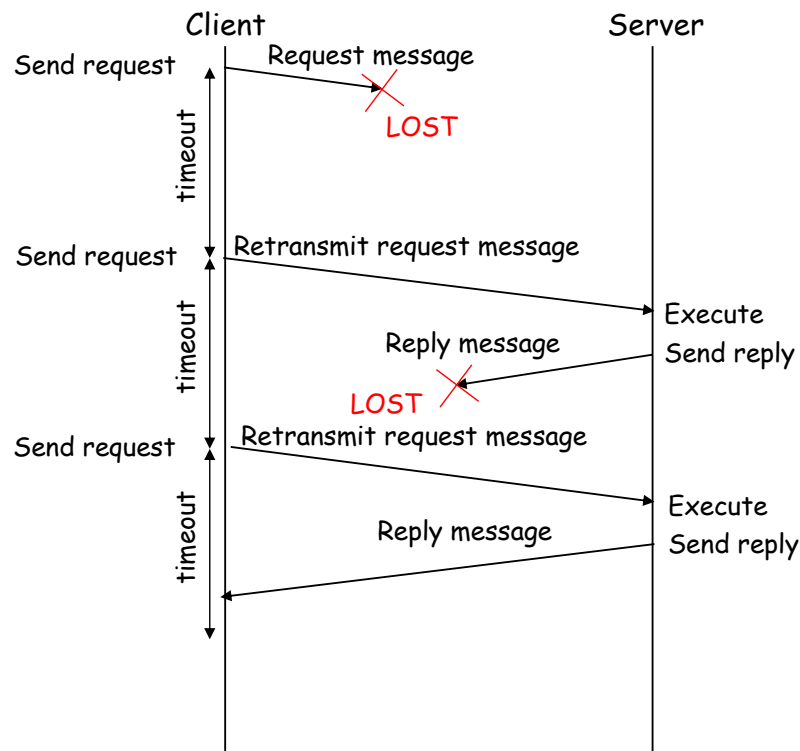
- No guarantee that the operation has been executed or not on server
- No action is taken to ensure the reliability of communication: no mechanism (RR1, DF, RR2) is used
- E.g., best-effort in UDP



At-least-once semantics

- The service, if executed, has been executed **at least once**
 - Could be several times, because of request duplication due to retransmissions
- Client uses RR1, server uses neither DF nor RR2
 - Server is not aware of duplicates
- Suitable for idempotent services (stateless server)
- Upon response receipt, client does not know how many times its request has been processed by server (at least once): client does not know about server status
 - The server may have executed the requested service but crashed before sending the response: when the timeout expires, the client resends the request and server executes the service again and sends the response to the client

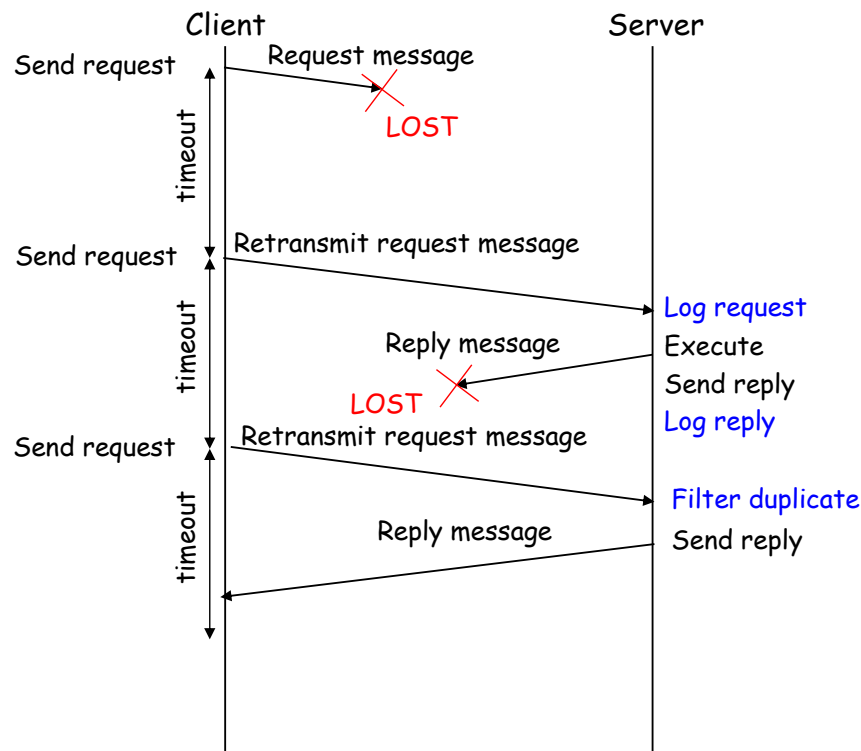
At-least-once semantics



At-most-once semantics

- Service, if performed, was carried out **at most once**
 - Client knows that if it receives the reply, it has been processed by server only once
 - In case of failure, no information (at-most-once: reply has been calculated at most once, but possibly also none)
- All basic mechanisms (RR1, DF, RR2) are used
 - Client retransmits request when timeout expires
 - Server maintains some **state** to identify duplicate requests and to not process the request more than once
- Suitable for any type of service
 - Even **not idempotent**
- No constraints on consequent actions
 - No coordination between client and server: in case of error, client does not know if server run the service, while server ignores if client knows that the service run
 - Possible inconsistency on the agreement between client and server

At-most-once semantics



At-most-once semantics: implementation

- Server detects duplicate requests and returns previous reply instead of re-running operation handler()
- How to detect duplicate request?
 - Client includes a unique ID (xid) with each request and uses same xid when re-sending
- Some at-most-once complexities
 - How to ensure xid is unique?
 - Server must eventually discard info about old requests: when is discard safe?
 - Can use sliding windows and sequence numbers
 - Can discard information older than maximum message lifetime
 - How to handle duplicate requests while original one is still executing?

Server:

```
if seen[xid]
  r = old[xid]
else
  r = handler()
  old[xid] = r
  seen[xid] = true
```

Exactly-once semantics

- Strongest but most difficult guarantees to implement in DS, especially large-scale ones
- Requires full agreement on the interaction
 - Service is run only once or it is not run at all: [all-or-nothing semantics](#)
 - If everything goes well: the service runs only once, duplicates are found
 - If something goes wrong: client or server knows if the service is run (once - all) or if it has not run (no time - nothing)
- Semantics with concordant knowledge of each other's state and without hypotheses on the maximum duration of the interaction protocol between client and server
 - No constraint on maximum duration: barely practical in a real system!

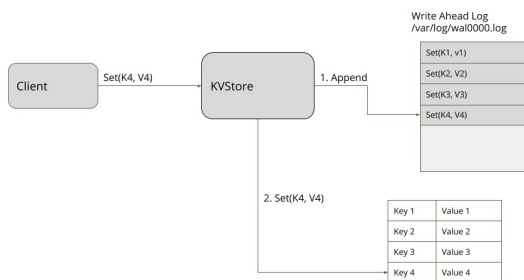
Exactly-once semantics: mechanisms

- Server-side basic mechanisms (RR1, DF, RR2) are not enough
- Additional mechanisms are required to tolerate server-side faults
 - [Transparent server replication](#)
 - [Write-ahead logging](#) (WAL)
 - [Recovery](#)
 - Mechanisms to recover from whatever state the failed server left behind and begin processing from a safe point
 - We will study distributed snapshot and state checkpointing

WAL pattern

- aka Commit log
- Goal
 - Provide durability guarantee by persisting every state change as a command to the append only log
- How
 - Each state change is stored as a log entry in a file on hard disk and the log is appended sequentially
 - The file can be read on every restart and the state can be recovered by replaying all the log entries

<https://martinfowler.com/articles/patterns-of-distributed-systems/wal.html>



```
public void put(String key, String value) {
    appendLog(key, value);
    kv.put(key, value);
}

private Long appendLog(String key, String value) {
    return wal.writeEntry(new SetValueCommand(key, value).serialize());
}
```

Summing up failure semantics

- At-least once and at-most once semantics are feasible and widely used in distributed systems
- We often choose the lesser of two evils, which is at-least-once semantics in most cases
 - At-least once semantics is also easier to scale

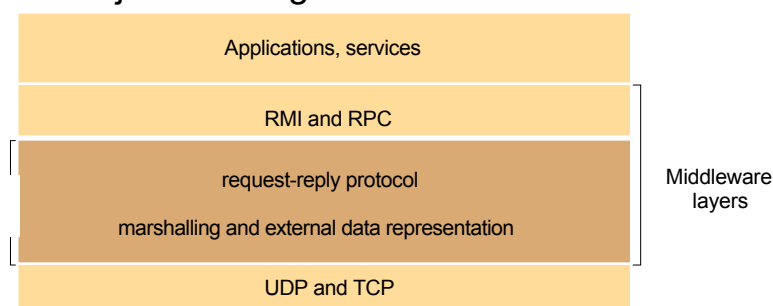
Distributed systems are all about trade-offs!

Distributed application programming

- You know **explicit** network programming
 - **Operating system construct** based on socket API and explicit management of message exchange
 - Used in most network applications (e.g. web browser, web server)
 - But distribution is not transparent and requires developer effort
- How to increase the abstraction level of distributed programming? By means of a **communication middleware** between OS and applications
 - Hide complexity of underlying layers
 - Free programmer from automatable tasks
 - Improve software quality by reusing known, correct and efficient solutions

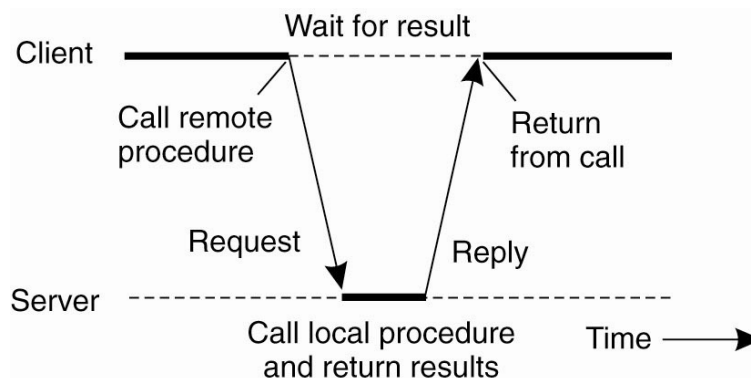
Distributed application programming

- **Implicit** network programming
 - **Language-level** construct
 - **Remote Procedure Call (RPC)**
 - Distributed app realized through procedure calls, but **caller** (client) and **callee** (server) are located on **different machines** and communication among them is hidden to programmer
 - **Remote method invocation (Java RMI)**
 - Distributed application in Java is realized by invoking **methods** of object running on a **remote machine**



Remote Procedure Call (RPC)

- Idea (by Birrel and Nelson, 1984): use client/server model to call procedures executed on other machines
 - Process on machine A calls procedure on machine B
 - Calling process on A is suspended
 - Called procedure is execute on B
 - Input and output parameters are transported into messages
 - No message passing is visible to programmer



Why RPC

- Used in many distributed systems, including cloud computing ones
- Developed and employed in many languages and frameworks, among which:
 - **C (Sun RCP)**
 - **Java (Java RMI)**
 - **Go**
 - **gRPC**
 - Remote Python Call ([RPyC](#))
 - Distributed Ruby ([DRb](#))
 - [Ice](#)
 - Microsoft .NET
 - [JSON-RPC](#)
 - CORBA

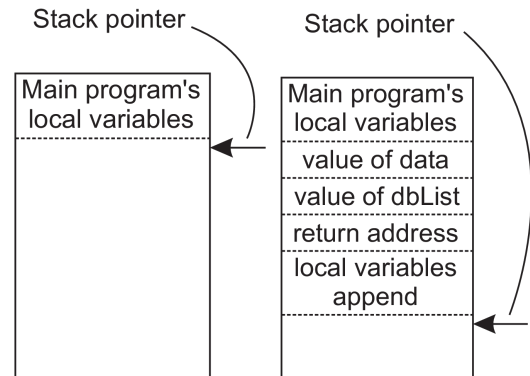
Our case studies

Local procedure call

- Example of local procedure call:

`newlist = append(data, dbList)`

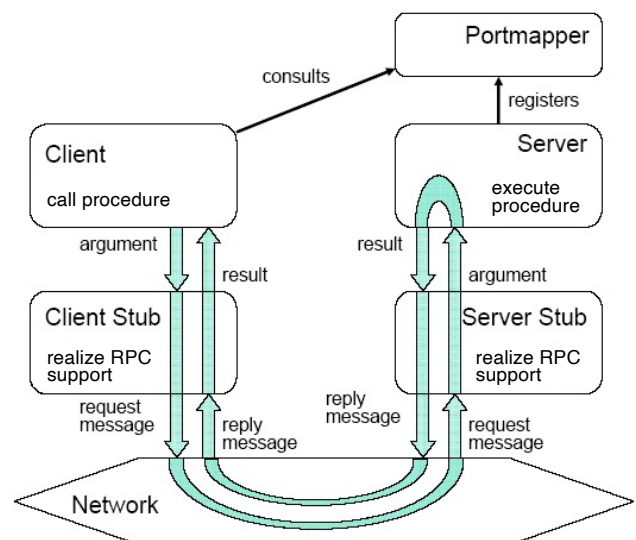
- Caller pushes to stack input parameters (data, dbList) and returns address
- When callee returns, control is back to caller



In case of RPC, how to make it appear to the developer that the call is local?

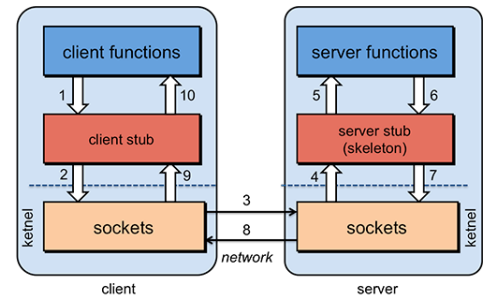
RPC: architecture

- Solution: create proxies (aka **stubs**)
- On the client side: the **client stub** has the service's interface
 - Client calls client stub that manages all the details: it packages parameters and calls the server
- On the server side: the **server stub** receives the request and calls the local procedure
- Goal: distribution transparency
 - Stubs are **automatically generated**
 - Developer focus is on application logic



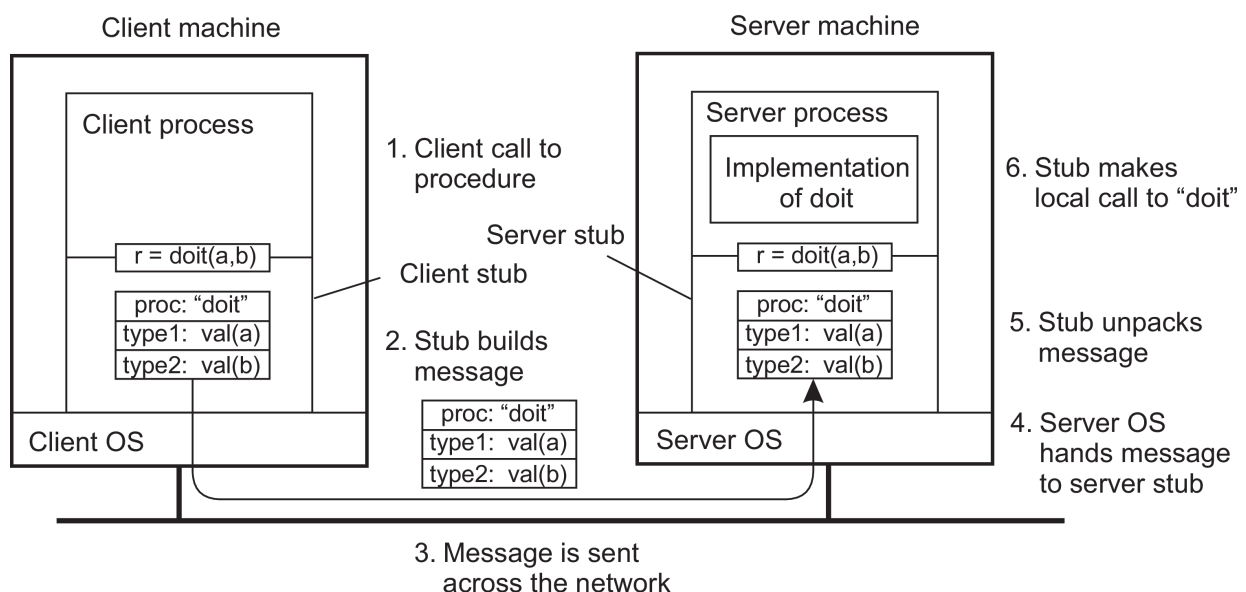
RPC: Basic steps

1. On client side, client calls a local procedure, called **client stub**
2. Client stub packs **request message** and call local OS
 - **Parameter marshaling**: arguments are converted from local to common format and packaged into a message
3. Client OS sends request message to remote OS
4. Remote OS gives request message to **server stub**
5. On server side, server stub **unpacks request message** and calls server as it was a local procedure
 - **Parameter unmarshaling**: arguments are extracted from message and converted from common to local format
6. Server executes local call and returns result to server stub
7. Server stubs packs **reply message** (marshals return value(s)) and calls OS
8. Server OS sends reply message to client OS
9. Client OS gives message to **client stub**
10. Client stub unpacks reply message (unmarshals return value(s)) and returns result to client



RPC basic steps: example

- Call remote procedure `doit(a,b)`



RPC middleware requirements

- **Exchange messages** to call procedure/invoke method, so to make it appear to the user that the call is local: we need to:
 - Identify request and reply messages, remote procedure/method
 - Pass parameters
- **Manage data heterogeneity**
 - Which data? Parameters, return value(s)
 - Marshaling vs. serialization:
 - **Serialization**: convert object into a sequence of bytes that can be sent over a network; serialization is used in marshaling
 - **Marshaling**: bundle parameters into a form that can be reconstructed (unmarshaled) by another process
- **Handle failures** due to distribution
 - During communication
 - User errors

RPC issues

- Issues to address in order to transparently execute procedure call
1. How to manage **heterogeneity** in data representation?
 - Client and server also need to agree on transport protocol for message passing: TCP, UDP, both?
 2. Client and server run on different hosts having their own address space: how to realize **parameter passing by reference**?
 3. When **failures** occur, what does the client know about the execution of the server?
 - Local procedure call: exactly-once
 - Remote procedure call: **at-least-once** or **at-most-once** (in most cases)
 4. How to **bind** to server, i.e., how do we locate the server endpoint?

Data heterogeneity

- Client and server may use different data representations
 - E.g., different character sets, byte ordering (little endian vs big endian), size of integers, floating point representation
- *General alternatives* (not only RPC) to handle heterogeneity in data representation:
 1. Specify encoding within the message itself
 2. Let message sender convert data into receiver encoding
 3. Convert data into standard encoding agreed between the parties
 - Sender: converts from local to standard
 - Receiver: converts from standard to local
 4. Let an intermediary convert between different encodings

Data heterogeneity

- Let's compare alternatives 2 and 3, assuming N distributed components that communicate among them
- #2: each component knows *all conversion functions*
Conversion is *faster*
X *Higher number* of conversion functions: $N*(N-1)$
- #3: all components agree on *standard encoding* for data representation and each component knows how to convert from local to common format and vice versa
Conversion is *slower*
X *Lower number* of conversion functions: $2*N$

Alternative 3 is the standard choice in RPC systems

Data heterogeneity: patterns

- Let's consider alternatives 3 and 4: how to realize?
- **Proxy**
 - Aim: support access (and location) transparency
 - Control access to an object using another proxy object
 - The proxy is created in the local address space to represent the remote object and offers the same interface of the remote object
- **Broker**
 - Aim: separate and encapsulate the details of communication from its functionality
 - Enables components to interact without handling remote concerns by themselves
 - Locates the server for the client, hides the communication details, etc.

Proxy (stub) is the standard choice in RPC systems

Who automatically generates stubs?

Parameter passing techniques

- **Call by value**
 - Parameter value is copied in a local isolated storage (usually stack)
 - The callee acts on copied data and changes will not affect the caller
- **Call by reference**
 - Reference (pointer) to parameter is copied into the stack
 - The callee acts directly on caller data
- **Call by copy-restore**
 - A somehow special case of call by reference: data is copied into the caller stack; when the procedure returns, the updated contents are copied back (restored)
 - Available in few programming languages (e.g., Ada, Fortran)

RPC parameter passing

- A reference is a memory address
 - Valid only in its context (local machine)
 - We need a pointerless representation
- Solution: simulate call by reference by using **call by copy-restore**
 - Client stub copies the pointed data in the request message and sends the message to server stub
 - Server stub acts on copy, using the address space of the receiver host
 - If the copy is modified, it will be then restored by client stub overwriting the original data
 - Size of data to be copied should be known
 - What happens if data contains a pointer?

Semantics of remote call/method

- Exactly once semantics is costly: most RPC systems implement weaker semantics
- **At-least-once semantics**: if the client gets a reply from the server, it means that remote call/method has been **executed at least once** by the server
- **At-most-once semantics**: if the client gets a reply from the server, it means that the remote call/method has been **executed at most once** by the server

Server binding

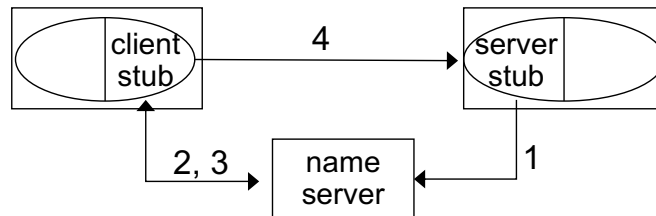
- Binding: how to locate the server endpoint, including the proper process (port or transport address) on it
 - In principle: can be static or dynamic
- **Static binding**
 - At design time: server address and other info are wired into code
 - Easy and no overhead, but lacks transparency and flexibility
- **Dynamic binding**
 - At run-time
 - Increased overhead, but gains transparency and flexibility
 - E.g., we can redirect requests in case of server replication
 - Try to limit overhead

Server binding: dynamic

- Two phases in client/server relationship
- **Naming**: static phase, before execution
 - Client specifies to whom it wants to be connected, using a unique name that identifies the service
 - Unique names are associated with operations or abstract interfaces and binding is made to the specific service interface
- **Addressing**: dynamic phase, during execution
 - Server effectively binds to client when client invokes service
 - Depending on middleware implementation, multiple replica servers can be looked for
 - Addressing can be *explicit* or *implicit*
 - **Explicit addressing**: client sends request using broadcast or multicast, waiting only for first reply

Server binding: dynamic

- **Implicit addressing**: there is a **name server** (aka **binder**, **directory service**, **registry service**) that registers services and manages a binding table
 - Service lookup, registration, update, and deletion



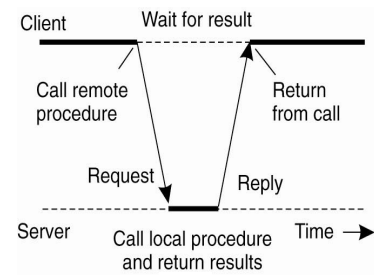
- Dynamic binding frequency
 - Each procedure call requires addressing
 - To reduce cost, binding result can be cached and re-used

RPC and OSI model

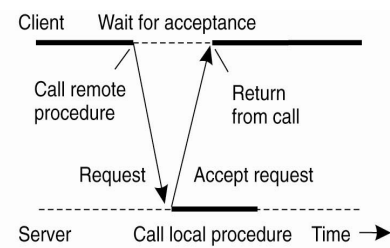
- Where is RPC in the OSI model?
 - Layer 5 (session): Connection management
 - Layer 6 (presentation): Marshaling/data representation
 - Uses the transport layer (layer 4) for communication (TCP/UDP)

More issues: Synchronous vs. asynchronous RPC

- Synchronous RPC: strict request-reply behavior
 - RPC call blocks client that waits for server reply
- Some RPC middleware supports **asynchronous RPC**
 - Client continues without waiting for server reply
 - Server can reply as soon as request is received and execute procedure later



Synchronous RPC



Asynchronous RPC

More issues: transparency

- Is RPC truly transparent? Can we really just treat remote procedure calls as local procedure calls?
 - **Performance**, **failures**, concurrent requests, replication, migration, ...
- Performance
 - RPC is slower ... a lot slower: why?
 - Local call: maybe 10 cycles = ~3 ns
 - RPC: 0.1-1 ms on a LAN => ~100K slower
 - Major source of overhead: context switching, copies, inter-process communication
 - In WAN: can easily be millions of times slower

More issues: transparency

- Failures
 - Different failures can occur
 - Client cannot locate server
 - Lost request messages
 - Server crashes
 - Lost reply messages
 - Client crashes

More issues: security

- Authenticate client? Authenticate server?
 - Is the client sending messages to the correct server or is the server an impostor?
 - Is the server accepting messages only from legitimate clients? Can the server identify user at the client side?
- Messages may be visible over network
 - Messages may be sniffed (and modified) while they traverse the network: do we need to hide them?
 - Has the message been accidentally corrupted or truncated while on the network?
- RPC protocol may be subject to replay attacks
 - Can a malicious host capture a message and retransmit it at a later time?

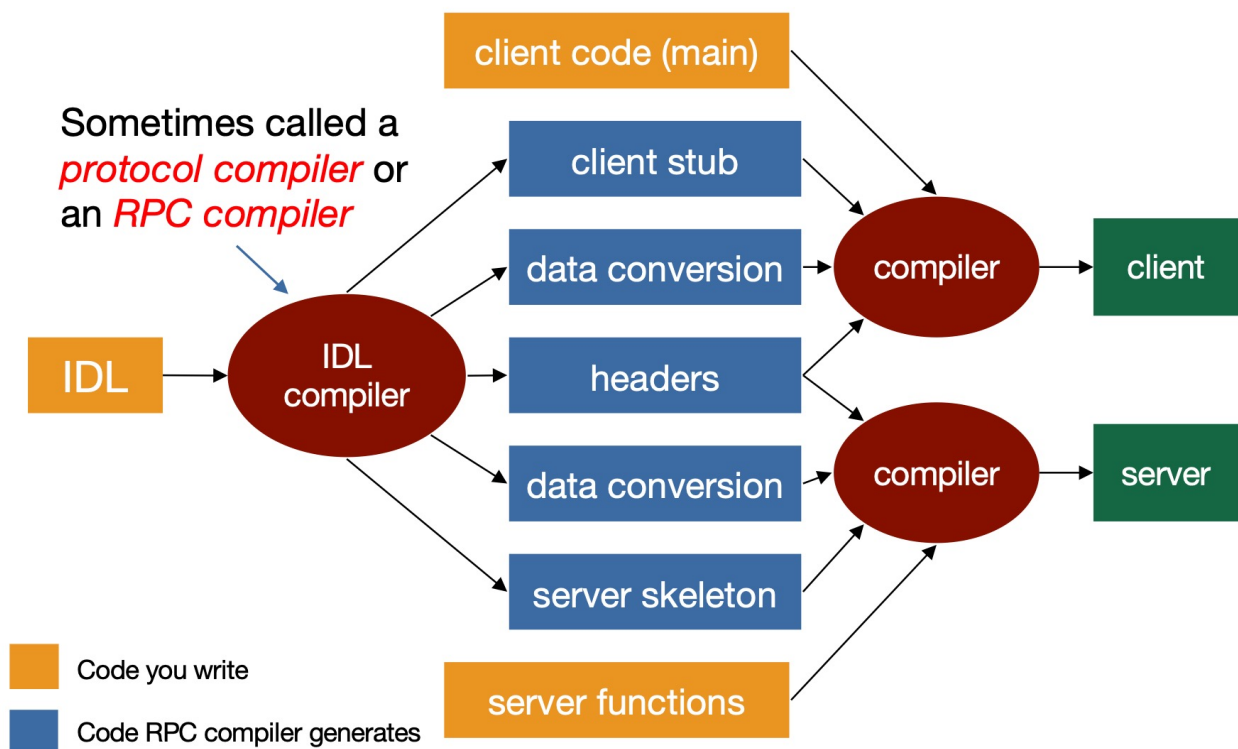
Programming with RPC

- Language support
 - Many programming languages have no language-level concept of remote procedure calls (e.g., C, C++)
 - These compilers will not automatically generate client and server stubs
 - Some languages have support that enables RPC (Java, Python, Haskell, Go, Erlang)
 - But we may need to deal with heterogeneous environments (e.g., Java communicating with a Python service)
- Common solution
 - **Interface Definition Language (IDL)**: describes remote procedures
 - Separate compiler that generates stubs (**pre-compiler**)

Interface Definition Language (IDL)

- Allow programmer to specify remote procedure interfaces (*names, parameters, return values*)
- IDL compiler can use this to generate client and server stubs
 - Marshaling code
 - Unmarshaling code
 - Network transport routines
- Conform to defined interface
 - An IDL looks similar to function prototypes

IDL and RPC compiler



RPC case studies

- Sun RPC
- Java RMI
- Go
- gRPC

Implementing RPC: Sun RPC

- Example of first generation RPC
- Implementation and RPC middleware provided by Sun Microsystems: Open Network Computing (ONC) RPC, aka Sun RPC
 - Basic and widely used implementation
 - RFC 1831 (1995), RFC 5531 (2009)
- ONC is a suite of products including:
 - eXternal Data Representation (XDR) as IDL
 - Remote Procedure Call GENerator (RPCGEN): IDL compiler to automatically generate client stub and server stub
 - Port mapper: service to bind client to server
 - Network File System (NFS): distributed file system

How to define RPC program

- Two descriptive parts written in XDR and grouped in a file with extension .x
 - In our example: square.x
 - 1. **Definition**: specifics of procedures (services) that is, identify procedures and their parameters' data types
 - 2. **XDR definitions**: definitions of parameters' data types (if not defined in XDR)
- Example: remote procedure to calculate square of integer number

Square example: define remote procedure

```
struct square_in {          /* input (argument) */
    long arg1;
};
struct square_out {         /* output (result) */
    long res1;
};
program SQUARE_PROG {
    version SQUARE_VERS {
        square_out SQUAREPROC(square_in) = 1; /* procedure number = 1 */
    } = 1;                                     /* version number */
} = 0x31230000;                          /* program number */
```

Code: square.x

Let's define the remote procedure SQUAREPROC

- Each procedure has only **one input parameter and one output parameter**
- Identifiers are written in **uppercase**
- Each procedure is associated with a **procedure number** which is unique within the RPC program (in the example 1)

How to implement RPC program

- Programmer has to develop:
 - **Client program**: implements `main()` and the logic needed to find the remote procedure and bind to it (example: `square_client.c`)
 - **Server program**: implements the remote procedures provided by RPC server (example: `square_server.c`)
- Note: programmer does not write server-side `main()`
 - Who calls the remote procedure on server side?

Square example: local procedure

- Let's first consider a standard solution for the **local** procedure

```
#include <stdio.h>
#include <stdlib.h>

struct square_in { /* input (argument) */
    long arg;
};
struct square_out { /* output (result) */
    long res;
};
typedef struct square_in square_in;
typedef struct square_out square_out;

square_out *squareproc(square_in *inp) {
    static square_out out;

    out.res = inp->arg * inp->arg;
    return(&out);
}
```

Code: square_local.c

Square example: local procedure

- Local** procedure (*continue*)

```
int main(int argc, char **argv) {
    square_in in;
    square_out *outp;

    if (argc != 2) {
        printf("usage: %s <integer-value>\n", argv[0]);
        exit(1);
    }
    in.arg = atol(argv[1]);

    outp = squareproc(&in);
    printf("result: %ld\n", outp->res);
    exit(0);
}
```

Which changes in case of remote procedure?

Square example: remote procedure

- Code of remote procedure is *almost equal* to local procedure

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "square.h" /* generated by rpcgen */
square_out *squareproc_1_svc(square_in *inp, struct svc_req
    *rqstp) {
    static square_out out;

    out.res1 = inp->arg1 * inp->arg1;
    return(&out);
}
```

Code: server.c

- Notes:
 - Input and output parameters use pointers
 - Output parameter must be a pointer to a static variable (i.e., global memory allocation) so that pointed area exists when procedure returns
 - Name of RPC procedure changes slightly (we add _ suffixed by version number and _svc, e.g., _1_svc), all in lowercase

Square example: client

- Launch client with remote hostname and integer value; it calls the remote procedure

```
#include <stdio.h>
#include <rpc/rpc.h>
#include "square.h" /* generated by rpcgen */
int main(int argc, char **argv) {
    CLIENT *clnt;
    char *host;
    square_in in;
    square_out *result;
    if (argc != 3) {
        printf("usage: client <hostname> <integer-value>\n");
        exit(1);
    }
    host = argv[1];
    clnt = clnt_create(host, SQUARE_PROG, SQUARE_VERS, "tcp");
```

Code: client.c

CLIENT *clnt_create(char *host, unsigned long prog, unsigned number vers, char *proto)

Square example: client

```
if (clnt == NULL) {
    clnt_pcreateerror(host);
    exit(1);
}
in.arg1 = atol(argv[2]);
if ((result = squareproc_1(&in, clnt)) == NULL) {
    printf("%s", clnt_sperror(clnt, argv[1]));
    exit(1);
}
printf("result: %ld\n", result->res1);
exit(0);
}
```

Square example: client

- **clnt_create()**: creates client transport manager to handle communication with remote server
 - TPC or UDP, default timeout for request retransmission
- Client must know:
 - Remote server hostname
 - Info to call remote procedure: program name, version number and remote procedure name
- To call remote procedure:
 - Procedure name changes slightly: we add `_` followed by version number (name in lowercase)
 - Two input parameters:
 - The first is the effective input parameter
 - The latter is the client transport manager
 - Client gets *pointer* to result
 - Allows it to identify failed RPC (null return)
- Handling of errors that occur during remote call
 - **clnt_pcreateerror()** and **clnt_perror()**

Basic steps to program with SUN RPC

1. Define the RPC interface: procedures and data types (if needed) → writes `square.x`
2. Compile `square.x` with `rpcgen`, which generates client and server stubs and XDR routines to convert data to XDR format
 - `square_clnt.c`, `square_svc.c`, `square_xdr.c`, `square.h`
3. Write client program and server functions (`client.c` and `server.c`), compile all source files (client and server, client and server stubs, and conversion routines) and link object files
4. When you start server, it publishes services
 - Remote procedures are registered with name server (port mapper or `rpcbind`)
5. When you start client, it finds the service endpoint through port mapper

Valeria Cardellini - SDCC 2021/22

64

Sun RPC features

- A program typically contains **multiple remote procedures**
 - **Multiple versions** for each procedure
 - **Single input and output parameter** (call by copy-restore)
- **Mutual exclusion** guaranteed by server: by default, no concurrency on server side
 - Sequential server: only one call can be executed at one time
 - Multi-threaded server (no Linux): `rpcgen` with `-M` and `-A` options
- Client is **synchronously blocked** waiting for server reply
- **At-least-once** semantics
 - Request retransmission after a time-out interval expires
 - UDP as default transport protocol

Valeria Cardellini - SDCC 2021/22

65

eXternal Data Representation (XDR)

- Sun RPC uses a **standard transfer format** called XDR to handle data heterogeneity
 - Standard for the description and encoding of machine-independent data (RFC 4506)
- Built-in XDR conversion functions for:
 - Predefined primitive types
 - E.g., `xdr_bool()`, `xdr_char()`, `xdr_int()`
 - Predefined structured types
 - E.g., `xdr_string()`, `xdr_array()`
- XDR is a **binary** format using *implicit typing*
 - *Implicit typing*: only values are transmitted, not data types or parameter info

Definition of file.x

- First part of file
 - XDR definition of **constants**
 - XDR definition of data types of input and output parameters for all data types for which there is no corresponding built-in XDR function
- Second part of file
 - XDR definition of **procedures**
 - Example in `square.x`: `SQUAREPROC` is the procedure name, procedure number 1, version 1 of program number `0x31230000`
 - According to Sun RPC:
 - Procedure number 0 is reserved for `NULLPROC`
 - Each procedure has a single input and output parameter
 - Identifiers for program, version and procedure are in uppercase

Sun RPC: server binding

- Procedure must be registered before being called
 - Unique triple given by: program number (*prognum*), version number (*versnum*), procedure number
 - Transport *protocol* must also be specified
 - Client must know the port number on which the server responds
- RPC server registers the RPC program in the *port map*
 - Dynamic table of RPC services on that host machine
 - Each table line contains the triple {*prognum*, *versnum*, *protocol*} and port
 - Procedure number is not specified: all the procedures within a program share the same transport manager
- The port map table is managed by only one process (called *port mapper*) for each host

Port mapper (rpcbind)

- Port mapper (rpcbind) listens on port 111
- At startup, server stub registers the RPC services it offers on port mapper
 - prognum, versnum, protocol and port*
- Client stub contacts port mapper to find out the corresponding port before invoking the remote procedure
- Port mapper registers the services and supports:
 - Insert a service
 - Delete a service
 - Lookup for service port
 - List registered services

Port mapper (rpcbind)

- To list all the RPC programs on a given host:
`rpcinfo -p hostname`

```
>$ rpcinfo -p
```

program	vers	proto	port	
100000	4	tcp	111	rpcbind
100000	4	udp	111	rpcbind
824377344	1	udp	59528	
824377344	1	tcp	49311	

824377344 (= 0x31230000) is the program number in square.x

- Available RPC programs are listed in `/etc/rpc`

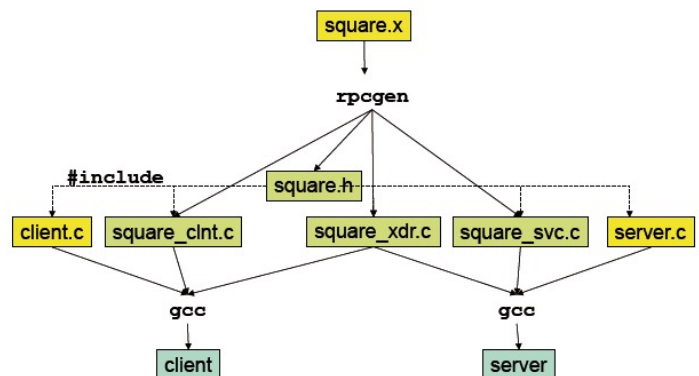
SUN RPC: Development process

Given a service specification

- Written in XDR: square.x

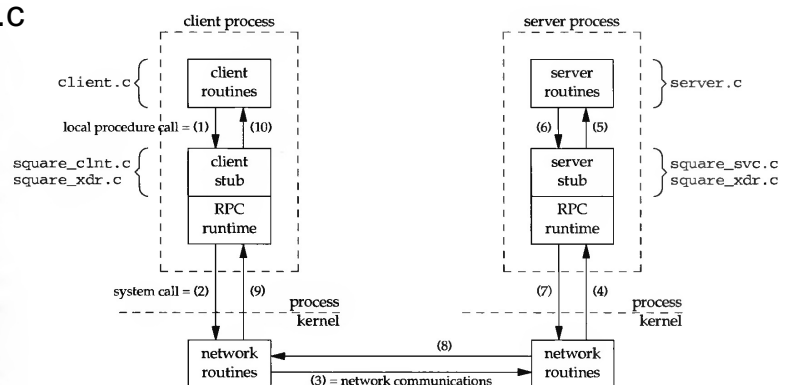
rpcgen generates

- Header: square.h
- Client stub: square_clnt.c
- Server stub: square_svc.c
- XDR routines: square_xdr.c



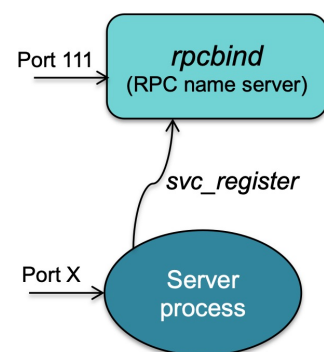
Developer writes

- Client program: client.c
- Server program: server.c



What goes on in the system: server side

- In `main()` server stub creates a socket and binds any available local port to it
- Calls `svc_register`, RPC library function
 - To register procedures with port mapper
 - Associates the specified program and version number pair with the specified dispatch routine
- Then waits for requests by calling `svc_run`, again in RPC library
 - `svc_run` invokes service procedures in response to RPC call messages

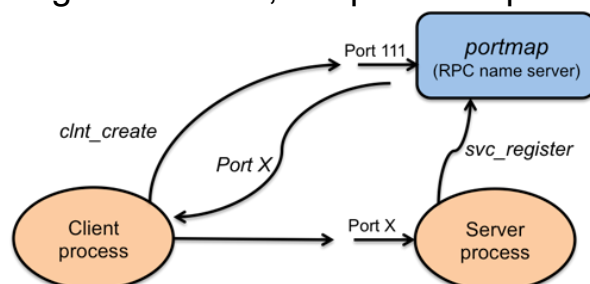


Valeria Cardellini - SDCC 2021/22

72

What goes on in the system: client side

- When we start the client program, `clnt_create` contacts port mapper on server side to find the port for that interface
 - Early binding: done once, not per each procedure call



- Client stub manages communication
 - Request timeout
 - Marshaling from local representation to XDR format and unmarshaling from XDR format to local representation

Valeria Cardellini - SDCC 2021/22

73

Examples in SUN RPC

- Examples of RPC programs
http://www.ce.uniroma2.it/courses/sdcc2122/prog_rpc
 - square: find square of an integer
 - Let's examine client stub and server stub code
 - echo: repeat a sequence of characters
 - avg: find average value of a sequence of real numbers
 - rls: print listing of a remote directory
 - XDR routines in dir_xdr.c are automatically generated from the data types defined in dir.x: they allow to convert from local format to XDR and vice versa

Second generation of RPC

- In the 1990s second generation of RPC
- Support for object oriented languages: distributed objects
 - Microsoft DCOM
 - CORBA
 - **Java RMI**

Java RMI: motivations

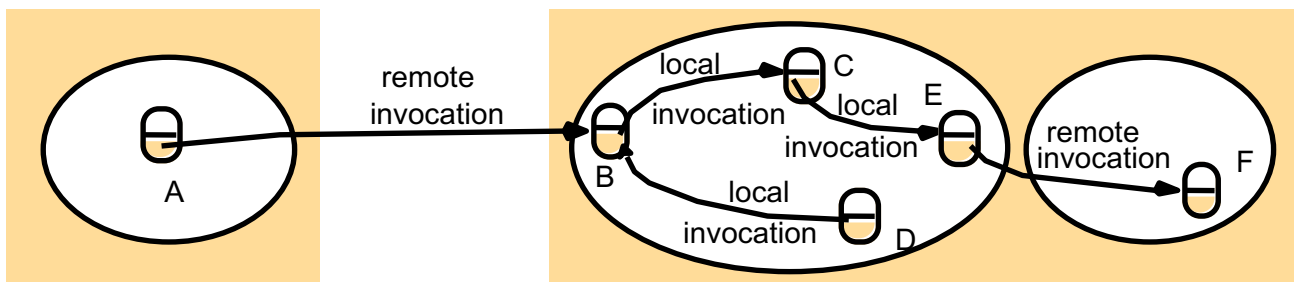
- Goal: extend RPC concepts to distributed objects
 - Java RMI (**Remote Method Invocation**): RPC in Java
 - Allows programmer to create distributed applications where methods of remote objects can be invoked from other JVMs
 - Conceptually similar to RPC but supports the semantics of object invocation in different address spaces
- Java RMI: set of tools, policies and mechanisms that allow a Java application running on a host to invoke methods of a remote object running on another host
 - Goal: access transparency (local and remote invocations are as much as possible the same)
 - But distribution transparency is not yet complete

Java RMI references

- “Trail: RMI”, The Java Tutorials
<http://docs.oracle.com/javase/tutorial/rmi/>
 - Note: JDK 8 (current release JDK 17)
- Java Remote Method Invocation Specification
<https://docs.oracle.com/en/java/javase/17/docs/specs/rmi/>

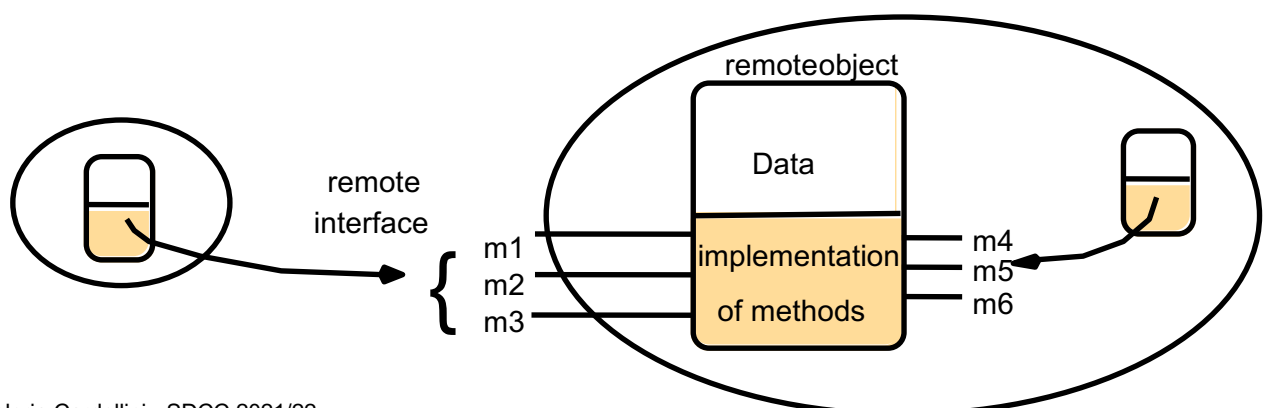
Java RMI basics

- Reference to remote object is created locally, but object is active on remote host
- Client invokes remote method through local reference
- Which are the differences with respect to invoking a local method?
 - Performance, reliability, communication, ...



Java RMI basics

- Separation between behavior definition (**interface**) and behavior implementation (**class**)
- Logical separation between interface and object allows for their physical separation
- Remote interface: specifies which object methods can be invoked remotely
 - Object internal state is not distributed!

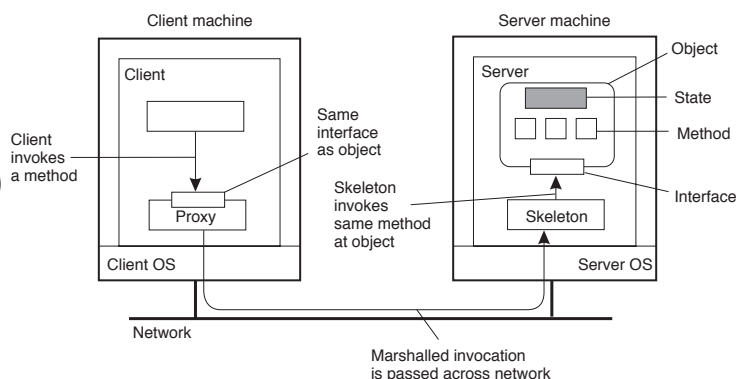


Java RMI basics

- When binding client with remote server object, the copy of the server interface (**stub**) is loaded into the client's space
 - Role similar to RPC client stub
- The request arriving at the remote object is handled by a client agent, which is local to the server (**skeleton**)
 - Role similar to RPC server stub
- Single working environment as a consequence of Java language, but underlying systems heterogeneity
 - Thanks to Java code portability

Stub and skeleton

- Like RPC, RMI exploits **proxy pattern**: two **proxies** (client-side **stub** and server-side **skeleton**) hide the application distributed nature to application layer
 - **Stub**: client-side proxy for the remote object; communicates method invocations on remote objects to server
 - **Skeleton**: server-side proxy that calls the actual remote object implementation



- Automatic stub generation
 - Automatically built from Java files (differently from Sun RPC rpcgen)

Serialization/deserialization

- (De)serialization directly supported by Java
 - Thanks to bytecode, no need to (un)marshal as in RPC, but data is (de)serialized using language-level features
- **Serialization**: transforms the object into a sequence of bytes
 - `writeObject` method on an output stream
- **Deserialization**: decodes a sequence of bytes and builds a copy of the original object
 - `readObject` method from an input stream
- Stub and skeleton use these two features to exchange input and output parameters with the remote host

Serialization/deserialization

- **Example of serializable Record object**

```
Record record = new Record();
FileOutputStream fos = new FileOutputStream("data.ser");
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(record);

FileInputStream fis = new FileInputStream("data.ser");
ObjectInputStream ois = new ObjectInputStream(fis);
record = (Record)ois.readObject();
```
- **Only instances of serializable objects can be used, that is:**
 - implement `Serializable` interface
 - contain only serializable objects (or object references)
- **Upon deserialization, a copy of the object will be recreated using its class file and the received information**
 - Class file must be available locally

Marshaling versus serialization

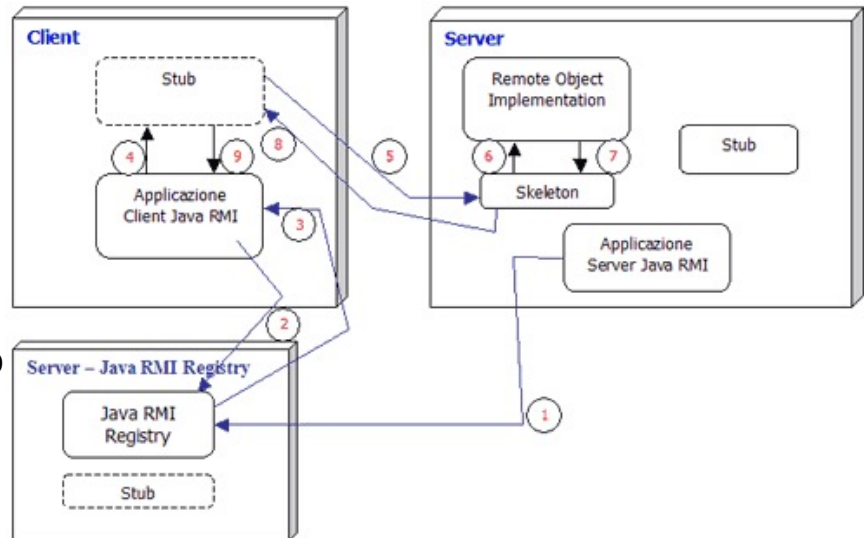
- Loosely synonymous but semantically different
- **Marshaling**: activity by which a stub converts local application data into network data (e.g., using XDR) and packages network data into packets for transmission
- **Serialization**: activity by which the state of an object is converted into a byte stream so that byte stream can be converted back into a copy of the object
- Difference becomes noticeable for objects
 - Objects have a codebase that also needs to be marshaled
 - Java serialization relies on codebase being present at receiver
- In Python (*pickle* module) marshaling and serialization are considered the same, but not in Java

Interaction between stub and skeleton

- Steps for communication
 1. Client **obtains a stub instance** (*how?*)
 2. Client **invokes methods** on the stub
 - Remote invocation syntax is identical to local one
 3. **Stub serializes** the information needed for the invocation (method's ID and parameters) and **sends** them to skeleton in a message
 4. **Skeleton receives** the message and **deserializes** received data, **invokes** the call on the object that implements the server (*dispatching*), **serializes** the return value and **sends** it to stub in a message
 5. **Stub deserializes** the return value and **returns** the result to the client

RMI registry

- **RMI Registry:** binder for Java RMI (port 1099)
- Allows server to publish service and allows client to obtain the proxy to access it



- RMI URL: starts with `rmi:` and contains the hostname (optional), the port number (optional) and the name of the remote object
- Some limit: no location transparency, no security management

Server-side steps

- Realize *server-side* remote components
 1. **Behavior definition:** **interface** that
 - is **public** so that it can be used by any class in any package
 - extends `java.rmi.Remote` so that its methods can be invoked by other JVMs
 - each remote method must declare that it may the remote exception `java.rmi.RemoteException` to indicate that either a communication failure or a protocol error has occurred
 2. **Behavior implementation:** **class** that
 - Implements the remote interface
 - Extends `java.rmi.UnicastRemoteObject`
 - **unicast:** the remote object is non-replicated
 - In general each class should at least:
 - Declare the remote interface being implemented
 - Define the constructor for the remote object
 - Provide an implementation for each remote method in the remote interface

Server-side and client-side steps

- Realize *server-side* remote components
 3. Server code:
 - Create an instance of the remote object
 - Register the remote object with the RMI registry, using **bind** or **rebind** (in `java.rmi.Naming`); **rebind** replaces any existing association
 - For security reasons, an application can only bind, unbind, or rebind remote object references with a registry running on the same host
- Realize *client-side* local components
 1. Obtains the reference to the remote object by invoking **lookup** on RMI registry
 2. Assigns it to a variable that has the remote interface as its type

Main steps to use Java RMI

- After coding:
 1. Compile classes
 2. Start RMI registry (`rmiregistry` command), which is launched in a separate process and has a standard structure and behavior
 - Alternatively, you can create your own local registry using `createRegistry` method in `java.rmi.registry`
 - RMI registry is local to the server for security reasons
 3. Start server
 4. Start client

Echo example: remote interface

- The interface extends the **Remote** interface

```
import java.rmi.Remote;  
import java.rmi.RemoteException;
```

- Each remote method

- Has to declare that it throws a **RemoteException**

- To handle communication failure or protocol error
- Remote method invocation **is not** completely transparent

```
public interface EchoInterface  
    extends Remote{  
    String getEcho(String echo)  
        throws RemoteException;  
}
```

- Returns only one result and has zero, one or more input parameters

- Parameter passing for remote method:

- **By value** in case of primitive data types (boolean, char, int, ...) or objects that implement the **java.io.Serializable** interface: serialization/deserialization managed by stub/skeleton
- **By reference** in case of **Remote** objects

Echo example: server

- The class that implements the server

- Has to extend the **UnicastRemoteObject** class
- **super()** method calls the class constructor **UnicastRemoteObject** which executes all the needed initialization to allow the server to wait for service requests and serve them
- Has to implement all the methods declared in the interface

```
public class EchoRMIServer  
    extends UnicastRemoteObject  
    implements EchoInterface{  
    //Costruttore  
    public EchoRMIServer()  
        throws RemoteException  
    { super(); }  
    // Implement the remote method declared  
    // in the interface  
    public String getEcho(String echo)  
        throws RemoteException  
    { return echo; }  
    public static void main(String[] args) {  
        // Service registration  
        try  
        { EchoRMIServer serverRMI =  
            new EchoRMIServer();  
            Naming.rebind("EchoService", serverRMI); }  
        catch (Exception e)  
        {e.printStackTrace(); System.exit(1); }  
    }  
}
```

Echo example: server

- In **main** the server object instance is created; then, it is ready to accept remote requests
- The RMI registry local to the server registers the services
 - **bind/rebind** methods in class **Naming** (**rebind** replaces an already existing binding)
 - Makes as many bind/rebind as the server objects to register, each one identified with a logical name
- Registering service provided by the RMI registry
 - Accepts bind/rebind requests only by the local registry

```
public class EchoRMIServer
    extends UnicastRemoteObject
    implements EchoInterface{
    //Costruttore
    public EchoRMIServer()
        throws RemoteException
    { super(); }
    // Implement the remote method declared
    // in the interface
    public String getEcho(String echo)
        throws RemoteException
    { return echo; }
    public static void main(String[] args) {
        // Service registration
        try
        { EchoRMIServer serverRMI =
            new EchoRMIServer();
            Naming.rebind("EchoService", serverRMI); }
        catch (Exception e)
        {e.printStackTrace(); System.exit(1); }
    }
}
```

Echo example: client

- Services used exploiting an interface variable, obtained by sending a **lookup** request to the RMI registry
- Lookup of a remote reference
 - Namely a stub instance of the remote object (using a lookup and assigning it to a interface variable)
- Remote method invocation
 - Synchronous blocking call using the parameters declared in the interface

```
public class EchoRMIClient
{
    // Start RMI client
    public static void main(String[] args)
    {
        bufferedReader stdIn =
            new BufferedReader(
                new InputStreamReader(System.in));
        try
        {
            // Connect to remote RMI service
            EchoInterface serverRMI = (EchoInterface)
                Naming.lookup("EchoService");
            // Interact with user
            String message, echo;
            System.out.print("Message? ");
            message = stdIn.readLine();
            // Invoke remote service
            echo = serverRMI.getEcho(message);
            System.out.println("Echo: "+echo+"\n");
        }
        catch (Exception e)
        {e.printStackTrace(); System.exit(1); }
    }
}
```

See code on course site

Echo example: compile and run

- Server side
 1. Compile interface source file and server class

```
javac EchoInterface.java
javac EchoRMIServer.java
```
 2. Start RMI registry and server

```
rmiregistry [registryPort]
java EchoRMIServer
```
- Client client
 1. Compile client class

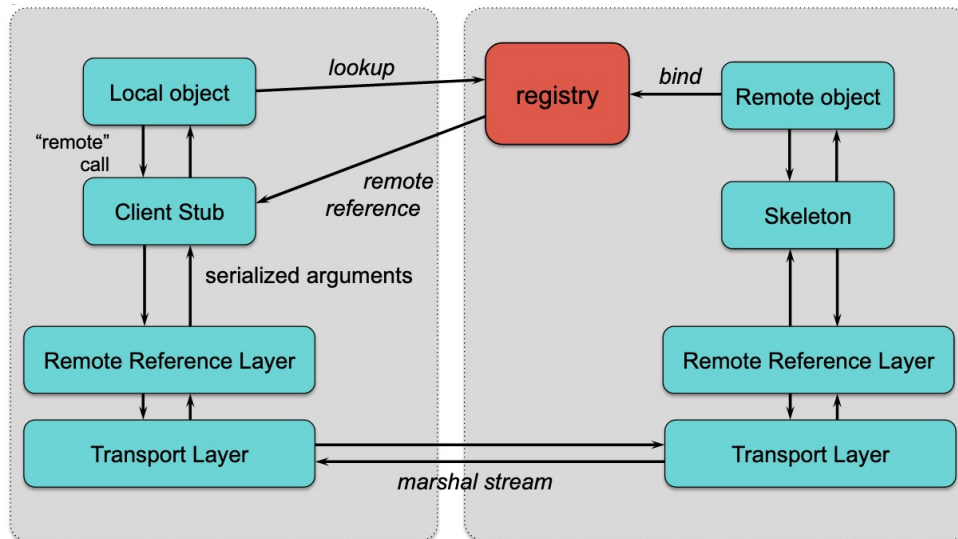
```
javac EchoRMIClient.java
```
 2. Start client

```
java EchoRMIClient
```

Java RMI: parameter passing

- Local method invocation:
 - By value: primitive types
 - By reference: all Java objects
- Remote method invocation:
 - **By value: primitive data types and serializable objects**
 - **Serializable objects:** object whose location is not relevant to the state are passed by value: the instance is serialized, sent to the destination and deserialized to build a local copy
 - **By remote reference: remote objects via RMI**
 - Objects whose utility is bounded to the location in which they run (the server) are passed by remote reference: their stub gets serialized and dispatched to the other peer
 - Each stub instance identifies a single remote object to which it refers through an identifier which is unique in the context of the JVM in which the target object exists

Java RMI: architecture



- Remote Reference Layer: manages remote references, parameters and stream-oriented connection abstraction
- Transport Layer
 - Manages connections between different JVMs
 - Can use different transport protocols, as long as they are connection-oriented (typically TCP)
 - Uses a proprietary protocol

Valeria Cardellini - SDCC 2021/22

96

Java RMI: concurrency support

- Methods of a remote object can be invoked concurrently by multiple clients

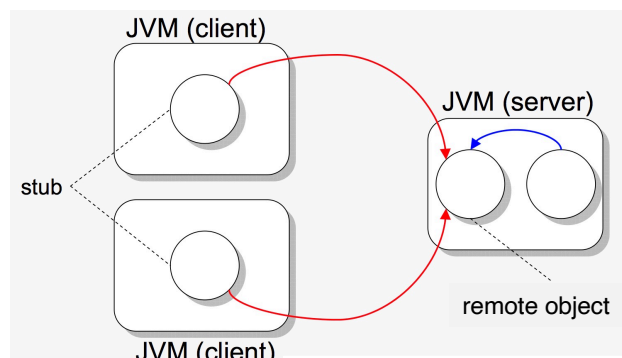
From Java RMI specification: *"Since remote method invocation on the same remote object may execute concurrently, a remote object implementation needs to make sure its implementation is thread-safe"*

<https://docs.oracle.com/en/java/javase/15/docs/specs/rmi/arch.html>

- To protect a remote method from "dangerous" concurrent accesses while guaranteeing thread safety, it must be defined as **synchronized**
- Example: remote method to increase a counter

Java RMI: distributed garbage collection

- How to delete remote objects that are no longer referenced by clients?
 - The remote object must know how many client stubs are using it
 - But network failures, client crashes, ...
- To address distributed garbage collection, RMI requires a high degree of coordination
 - Con: no large-scale apps



Valeria Cardellini - SDCC 2021/22

98

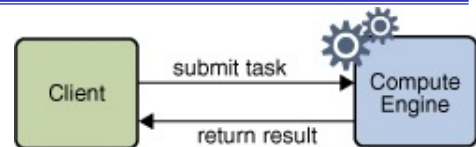
Java RMI: distributed garbage collection

- Within a JVM, Java uses reference counting and schedules objects for garbage collection when the reference count goes to zero
- Across JVMs, with RMI, Java supports two operations (**dirty** and **clean**) to realize a **lease-based garbage collection**
 - Client JVM periodically sends **dirty** message to server JVM when a remote object is in use
 - Dirty is refreshed based on lease time given by server
 - Client JVM sends **clean** message when there are no more local references to the object
 - If server JVM does not receive either **dirty** or **clean** before the lease time expires, object can be scheduled for deletion if it is not referenced by anyone
 - If a client does not renew the lease before it expires, server assumes that remote object is no longer referenced by that client

Valeria Cardellini - SDCC 2021/22

99

Example: compute engine



- Realize a compute engine
 - Remote object on server that takes tasks from clients, runs tasks, and returns any result
 - Tasks are defined by clients but executed on server machine
 - Powerful server machine or specialized hardware
 - Tasks can be arbitrary: the task class only needs to implement a particular interface
 - Compute engine downloads task code and runs it into its JVM
- Two interfaces to implement the compute engine
 - Remote interface Compute enables tasks to be submitted to the engine
 - Interface Task defines how the compute engine executes a submitted task

Code: <http://docs.oracle.com/javase/tutorial/rmi/>

Compute engine: remote interface and argument

- Compute engine's remote interface

```
package compute;
```

```
import java.rmi.Remote;
```

```
import java.rmi.RemoteException;
```

```
public interface Compute extends Remote {  
    <T> T executeTask(Task<T> t) throws RemoteException;  
}
```

- Task interface: type of parameter to executeTask method in Computeinterface

```
package compute;
```

```
public interface Task<T> {  
    T execute();  
}
```

Compute engine: server

```
package engine;

import java.rmi.RemoteException;
import java.rmi.registry.LocateRegistry;
import java.rmi.registry.Registry;
import java.rmi.server.UnicastRemoteObject;
import compute.Compute;
import compute.Task;

public class ComputeEngine implements Compute {
    public ComputeEngine() { super(); }
    public <T> T executeTask(Task<T> t) {
        return t.execute();
    }
}
```

Compute engine: server

```
public static void main(String[] args) {
    if (System.getSecurityManager() == null) {
        System.setSecurityManager(new SecurityManager());
    }
    try {
        String name = "Compute";
        Compute engine = new ComputeEngine();
        Compute stub =
            (Compute) UnicastRemoteObject.exportObject(engine,0);
        Registry registry = LocateRegistry.getRegistry();
        registry.rebind(name, stub);
        System.out.println("ComputeEngine bound");
    } catch (Exception e) {
        System.err.println("ComputeEngine exception:");
        e.printStackTrace();
    }
}
```

```
}
```


Compute engine: client

```
public class ComputePi {  
    public static void main(String args[]) {  
        if (System.getSecurityManager() == null) {  
            System.setSecurityManager(new SecurityManager());  
        }  
        try {  
            String name = "Compute";  
            Registry registry = LocateRegistry.getRegistry(args[0]);  
            Compute comp = (Compute) registry.lookup(name);  
            Pi task = new Pi(Integer.parseInt(args[1]));  
            BigDecimal pi = comp.executeTask(task);  
            System.out.println(pi);  
        } catch (Exception e) {  
            System.err.println("ComputePi exception:");  
            e.printStackTrace();  
        }  
    }  
}
```

Valeria Cardellini - SDCC 2021/22

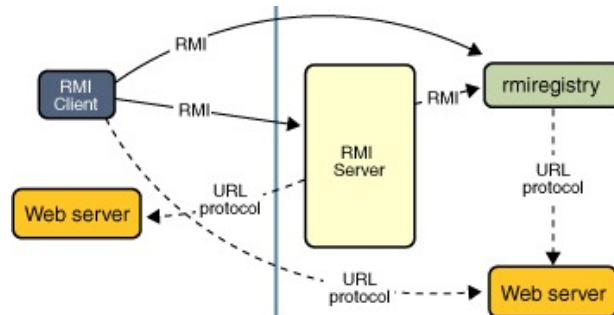
104

Compute engine: client's compute object

```
package client;  
  
import compute.Task;  
import java.io.Serializable;  
import java.math.BigDecimal;  
  
public class Pi implements Task<BigDecimal>, Serializable {  
    private final int digits;  
    public Pi(int digits) { this.digits = digits; }  
    // lot of stuff deleted ...  
    public BigDecimal execute() {  
        return computePi(digits);  
    }  
    // more stuff deleted ...  
}
```

Example: compute engine

- Java RMI can use dynamic class loading and security manager to transport Java classes safely
- In Compute engine example:
 - **Dynamic class loading** through web server



- Install **security manager** and define related security policy files
 - So that loaded classes can perform only granted operations
- <https://docs.oracle.com/javase/tutorial/rmi/running.html>

Comparing SUN RPC and Java RMI

- **SUN RPC**: process oriented, incomplete access transparency, no location transparency
- *Entities that can be requested*: operations or functions
- *Communication*: synchronous and asynchronous
- *Communication semantics* (default): at-least-once
- *Maximum duration and exceptions*: timeout for retransmission and error handling
- *Server binding*: port mapper on the server
- *Data presentation*: specific IDL (XDR) and automatic generation of client stubs and server stubs
- *Parameter passing*: by copy-restore
- Various extensions, including broadcast (multiple responses from multiple servers rather than just one) and security

Comparing SUN RPC and Java RMI

- **Java RMI**: object oriented, access transparency, no location transparency, distribution is not totally transparent (semantics of parameter passing, remote interface definition and remote exceptions)
- *Entities that can be requested*: methods of objects via interfaces
- *Communication*: synchronous
- *Communication semantics*: at-most-once
- *Maximum duration and exceptions*: error handling
- *Server binding*: RMI registry
- *Data presentation*: Java as IDL and automatic generation of stubs and skeletons
- Parameter passing : by value (primitive types and serializable objects), by reference in case of objects with remote interfaces (remote objects)

RPC in Go

- Let us analyze:
 - Main features of Go programming language
 - How Go supports RPC

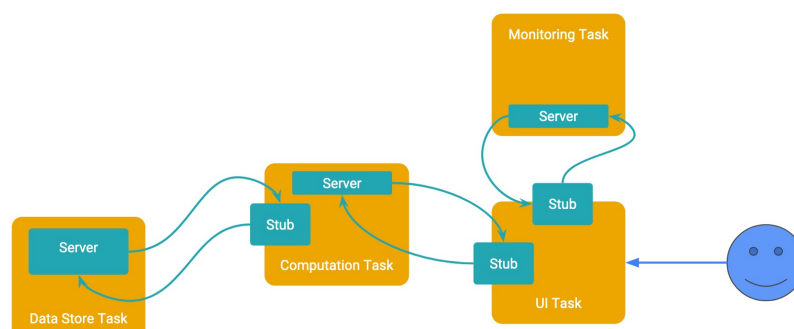
<http://www.ce.uniroma2.it/courses/sdcc2122/slides/Go.pdf>

Comparing RPC implementations

- How do SUN RPC, Java RMI and Go differ in terms of distribution transparency?

Motivation for new RPC middleware

- Large-scale distributed applications composed of **microservices**
 - Microservices architecture: building a software application as a collection of *independent*, *autonomous* (developed, deployed, and scaled independently), *business capability-oriented*, and *loosely coupled* services
 - Even multi-language (i.e. polyglot) development
 - Use communication predominantly structured as RPCs



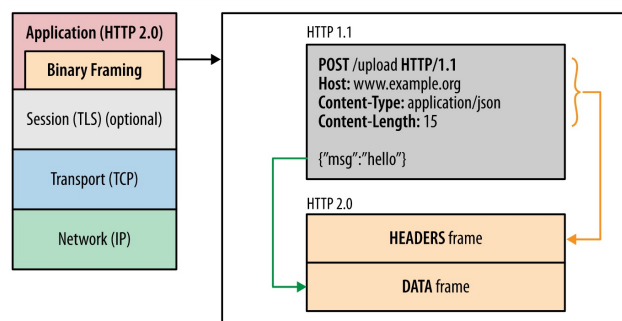
- High-performance, open source universal RPC framework <https://grpc.io/>
- Can run in any environment
 - Multi-language, multi-platform framework
- Main usage scenarios
 - Connect polyglot microservices that use request-response style communication
 - Connect mobile devices to backend services
 - Generate efficient client libraries
- Used by many companies and in many distributed systems
 - Google, IBM, Netflix, Dropbox, ..., [etcd](#), CockroachDB , ...
- Reference
 - Indrasiri and Kuruppu, "gRPC - Up and Running", O'Reilly, 2020

gRPC: Main features

- HTTP/2 for transport
- Protocol buffers as IDL
- Plus authentication, bidirectional streaming and flow control, blocking or non-blocking bindings, and cancellation and timeouts

gRPC: HTTP/2

- Transport over **HTTP/2**
 - Basic idea of gRPC: treat RPCs as references to HTTP objects
- HTTP/2: major revision of HTTP that provides significant performance benefits over HTTP 1.x
- HTTP/2 in a nutshell
 - **Binary framing layer**: HTTP/2 request/response is divided into small messages and framed in binary format, making message transmission efficient



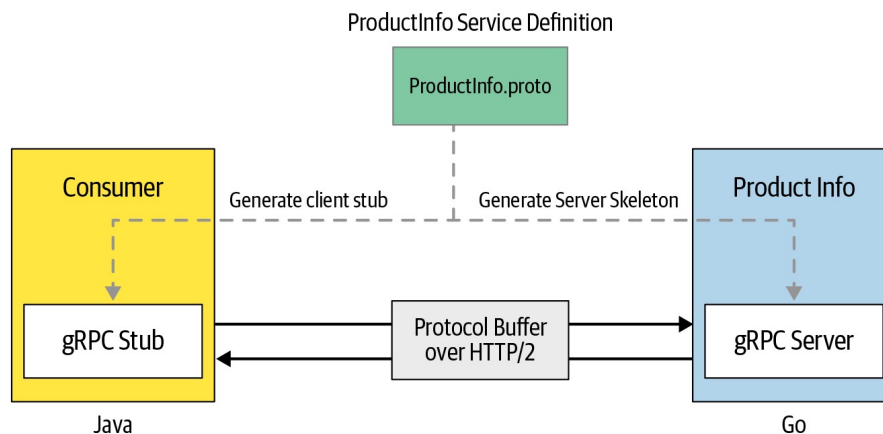
gRPC: HTTP/2

- HTTP/2 in a nutshell
 - From request/response messages to **streams**
 - *Stream*: bidirectional flow of bytes within an established connection, which may carry one or more messages
 - *Message*: complete sequence of frames that map to a logical request or response message
 - *Frame*: smallest unit of communication in HTTP/2, each containing a frame header, which at a minimum identifies the stream to which the frame belongs
 - Request/response **multiplexing** (usage of a single connection per client): allows for efficient use of TCP connections and avoids head-of-line blocking at HTTP level
 - Native support for **bidirectional streaming**
 - HTTP **header compression**: to reduce protocol overhead

<https://developers.google.com/web/fundamentals/performance/http2?hl=it>

gRPC: Protocol buffers

- gRPC can use **protocol buffers** as both its IDL to define the service interface and as its underlying message interchange format
 - Automatically generates client stubs and abstract server classes
- Based on usual **proxy pattern** (*stub* and *server*)



Valeria Cardellini - SDCC 2021/22

116

Protocol buffers

- Google's mature open-source mechanism for serializing structured data
- Binary data representation
- Strongly typed

```
message Person {  
  string name = 1;  
  int32 id = 2;  
  bool has_ponycopter = 3;  
}
```

- Data types are structured as messages
 - Each message is a small logical record of information containing a series of name-value pairs called fields
 - Fields have unique field numbers (e.g., `string name = 1`) that are used to identify the fields in the message binary format

Valeria Cardellini - SDCC 2021/22

117

Protocol buffers: example

- ProductInfo service interface (slide 114)

```
// ProductInfo.proto
syntax = "proto3";
package ecommerce;

service ProductInfo {
    rpc addProduct(Product) returns (ProductID);
    rpc getProduct(ProductID) returns (Product);
}

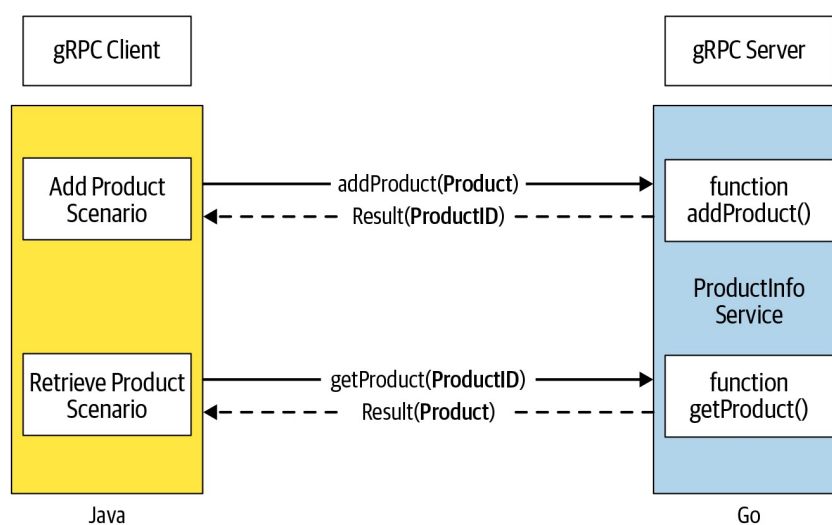
message Product {
    string id = 1;
    string name = 2;
    string description = 3;
}

message ProductID {
    string value = 1;
}
```

Valeria Cardellini - SDCC 2021/22

118

Client-server interaction: example



Valeria Cardellini - SDCC 2021/22

119

gRPC: basic steps

1. Define the service (collection of remote methods) and the message types that are exchanged between client and service in a .proto file using protocol buffers as IDL
2. Generate server and client code using protoc ([protocol buffer compiler](#)) in your preferred language(s) from your proto definition
 - Go: compile manually
 - Java: use build automation tools like Bazel, Maven, or [Gradle](#)
3. Use the gRPC API in your preferred language (e.g., Go, Java, Python) to write the service client and server

gRPC: greeting example in Go

- See [Quick start/](#) and [helloworld](#) example
1. Define the service ([helloworld.proto](#) file)

```
package helloworld;
```

```
// The greeting service definition.
```

```
service Greeter {
```

```
    // Sends a greeting
```

```
    rpc SayHello (HelloRequest) returns (HelloReply) {}
```

```
}
```

```
// The request message containing the user's name.
```

```
message HelloRequest {
```

```
    string name = 1;
```

```
}
```

```
// The response message containing the greetings
```

```
message HelloReply {
```

```
    string message = 1;
```

```
}
```

gRPC: greeting example

2. Compile the service definition:

```
$ protoc --go_out=. --go_opt=paths=source_relative \  
    --go-grpc_out=. --go-grpc_opt=paths=source_relative \  
    helloworld/helloworld.proto
```

- **Generated files:**

- `helloworld.pb.go`: contains all the protocol buffer code to populate, serialize, and retrieve request and response message types
- `helloworld_grpc.pb.go`: contains
 - An interface type (or *stub*) for clients to call with the methods defined in the Helloworld service
 - An interface type for servers to implement, also with the methods defined in the Helloworld service

gRPC: greeting example

3. Create the server: two parts

- a. Implement the service interface generated from the service definition: doing the actual “work” of the service

```
func (s *server) SayHello(ctx context.Context,  
    in *pb>HelloRequest) (*pb>HelloReply, error) {  
    ...  
}
```

- b. Run a gRPC server to listen for requests from clients and dispatch them to the right service implementation

```
lis, err := net.Listen("tcp", port)  
if err != nil {  
    log.Fatalf("failed to listen: %v", err)  
}  
s := grpc.NewServer()  
pb.RegisterGreeterServer(s, &server{})  
s.Serve(lis)
```

gRPC: greeting example

4. Create the client

- To call service methods, we first need to create a gRPC *channel* to communicate with the server using `Dial`
- Then we need a client stub to perform RPCs: we get it using `pb.NewGreeterClient` provided by the `pb` package generated from `.proto` file.

```
conn, err := grpc.Dial(address, opts...)
```

```
c := pb.NewGreeterClient(conn)
```

- Then we call the service method on the client stub: we create and populate a request protocol buffer object (`HelloRequest`) and also pass a context object which lets us change our RPC's behavior if necessary, such as timeout/cancel an RPC in flight

```
r, err := c.SayHello(ctx, &pb>HelloRequest{Name: name})
```

gRPC: greeting example

- As next step, we can update the gRPC service adding a new `SayHelloAgain()` method

1. Update `.proto` file
2. Regenerate gRPC code using `protoc`
3. Update server code to implement new method

```
func (s *server) SayHelloAgain(ctx context.Context, in
*pb>HelloRequest) (*pb>HelloReply, error) {
    log.Printf("Received: %v", in.GetName())
    return &pb>HelloReply{Message: "Hello again " +
in.GetName()}, nil
}
```

4. Update client code to call new method

```
r, err = c.SayHelloAgain(ctx, &pb>HelloRequest{Name: name})
if err != nil {
    log.Fatalf("could not greet: %v", err)
}
log.Printf("Greeting: %s", r.GetMessage())
```

gRPC: ProductInfo example

- Let's analyze the ProductInfo example
 1. Define the service
 2. Implement server in Go
 3. Implement client in Go
 4. Implement client/server in Java

See Go code on course web site

See Java code on [github](#)

gRPC: types of RPC methods

- Multiple types of methods can be defined in .proto file
- *Simple RPC*: client sends a request to server and waits for a response to come back

```
rpc SayHello (HelloRequest) returns (HelloReply) {}
```

- *Server-side streaming RPC*: client sends a request to server and gets a stream to read a sequence of messages back

```
rpc ListFeatures(Rectangle) returns (stream Feature) {}
```

- *Client-side streaming RPC*: client writes a sequence of messages and sends them to server

```
rpc RecordRoute(stream Point) returns (RouteSummary) {}
```

- *Bidirectional streaming RPC*: both sides send a sequence of messages using a read-write stream

```
rpc RouteChat(stream RouteNote) returns (stream RouteNote) {}
```

See [routeguide](#) example

gRPC: Weaknesses

- Limited browser support
 - Impossible to directly call a gRPC service from a browser because of lack of HTTP/2 support
 - [gRPC-Web](#) can be used to provide gRPC support in browser, but limited features (only simple RPC and limited server streaming)
- Non-human readable format
 - Protocol buffers is efficient to send and receive, but its binary format is not human readable
 - Developers need additional tools (e.g., gRPC command-line tool) to analyze Protobuf payloads on the wire, write manual requests, and perform debugging