Remote Procedure Call (RPC)

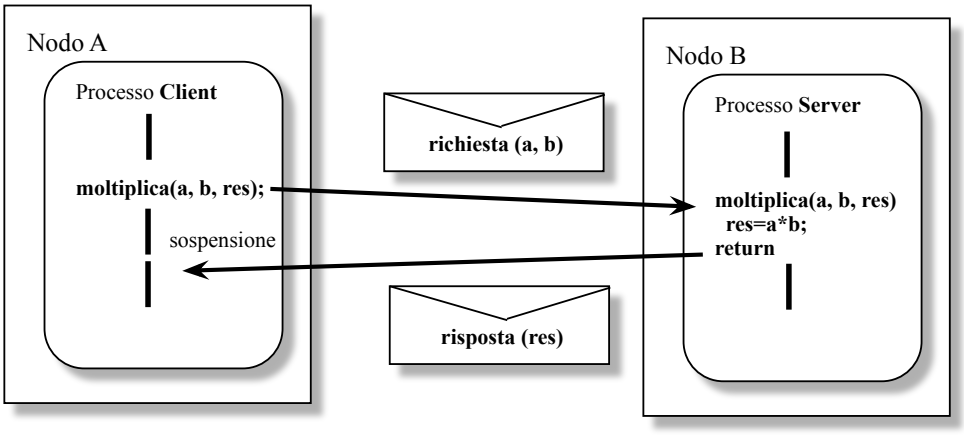
I modelli a scambio di messaggi hanno un **basso livello di astrazione** e caricano il programmatore di lavoro, per es., per costruire i messaggi, per trasformare tipi di dato in sistemi eterogenei, etc.

RPC è un modello per mascherare i dettagli di basso livello facilitando il compito dei programmatori.

Scopo di **RPC** è di far sembrare l'invocazione di un servizio remoto come se fosse una chiamata di procedura locale. I parametri dell'invocazione sono contenuti in un messaggio di richiesta, la risposta al servizio in un messaggio di risposta.

Tutti i linguaggi imperativi hanno chiamate a procedura, per cui RPC si integra bene con i tradizionali linguaggi di programmazione, e permette di uniformare programmi concentrati e distribuiti.

RPC



Il Client è sospeso fino al completamento della chiamata.

I parametri devono essere passati per valore, visto che i processi Client e Server non condividono lo spazio si indirizzamento (un puntatore del Client non ha significato nello spazio di indirizzamento del Server).

Differenze RPC e chiamata a procedura locale

Vi sono importanti differenze semantiche tra RPC e chiamata a procedura locale. In una chiamata remota:

- sono coinvolti due processi diversi, che:
  - **non condividono** lo spazio di indirizzamento
  - hanno vita **separata**
  - possono eseguire su macchine **eterogenee**
- si possono verificare **malfunzionamenti**, sia nei nodi, sia nell'interconnessione

Implementazioni RPC

Vi sono molte diverse implementazioni di RPC, con caratteristiche significativamente differenti, tra cui:

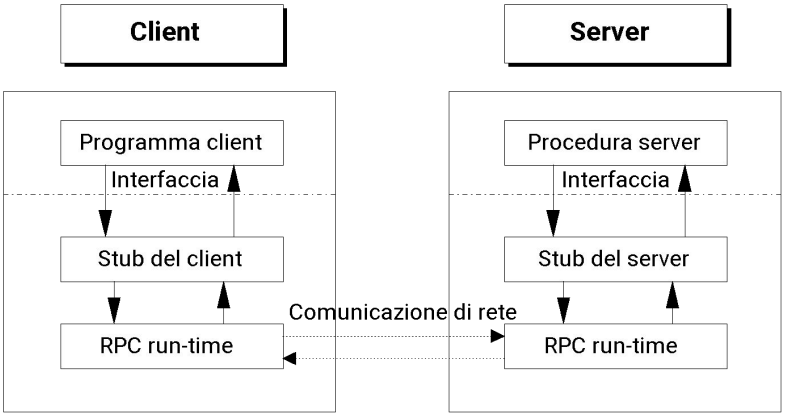
- Sun/ONC RPC, meccanismo di RPC piuttosto diffuso in ambiente Unix/Linux standardizzato da Open Network Computing e usato in NFS e NIS/YP
- DCE RPC, meccanismo di RPC alla base del Distributed Computing Environment (DCE) standardizzato da The Open Group
- MSRPC, implementazione Microsoft-specific di DCE/RPC usata in molti servizi di rete e middleware (tra cui DCOM) proprietari
- XML-RPC e SOAP, basati su standard Web (XML e HTTP)
- gRPC (Google Remote Procedure Call)

Attenzione a non confondere il meccanismo generale di RPC con una sua particolare implementazione.

Come si realizza RPC

Utilizzo di **procedure stub**:

- il **Client** invoca uno **stub** che si incarica del trattamento dei parametri e richiede al supporto il trasporto del messaggio di richiesta
- il messaggio di richiesta arriva a un **Server stub**, che estrae dal messaggio i parametri, invoca la procedura e al completamento del servizio rimanda il risultato al Client



RPC - le procedure stub

Un Client ha una procedura stub locale per ogni procedura remota che può chiamare.

Un Server ha uno stub locale per ogni procedura che può essere chiamata da un Client remoto.

Compiti degli stub:

- **marshalling** dei parametri, cioè impacchettare i parametri in un messaggio
- estrazione parametri dai messaggi
- **trasformare la rappresentazione** dei dati
- accedere alle primitive di comunicazione per spedire e ricevere messaggi

Gli stub sono **generati automaticamente** a partire dalla specifica dell'interfaccia.

Interface Definition Language (IDL)

Definizione di **linguaggi astratti** per la specifica del **servizio.**

IDL sono linguaggi usati per specificare le operazioni dell'interfaccia e i loro parametri.

Esempio di IDL (DCE RPC):

```
interface calcolatore
{
   void moltiplica([in] float a, [in] float b,
                   [out] float *res);
   void radice([in] float a, [out] float *res);
}
```

Una specifica IDL viene elaborata da un compilatore che produce sia il Client sia il Server stub.

I compilatori per IDL possono produrre stub in differenti linguaggi.

Binding tra Client e Server

Il binding lega l'interfaccia di RPC usata da un Client all'implementazione fornita da un Server.

Sono possibili diverse soluzioni (statico vs dinamico) per il binding tra il Client e il Server:

binding statico. Il binding viene eseguito a tempo di compilazione rimane inalterato per tutto il tempo di vita dell'applicazione.

binding dinamico. Una scelta dinamica consente di ridirigere le richieste sul gestore più scarico o quello in quel momento disponibile nel sistema. In teoria si potrebbe pensare di eseguire un nuovo binding per ogni richiesta tra Client e Server, nella pratica si usa lo stesso binding per molte richieste e chiamate allo stesso Server.

Fasi del Binding

È opportuno evidenziare due diverse fasi del binding:

- **nome servizio**, il Client deve specificare a chi vuole essere connesso, in termini del nome identificativo del servizio (è la fase di **naming**, fase statica). Ogni servizio, ogni operazione, deve essere identificata da un nome unico.

- **indirizzo del Server**, il Client deve essere realmente collegato al servitore che fornisce il servizio nel sistema al momento della invocazione (è la fase di **addressing**, fase dinamica)

Fasi del Binding

In RPC il binding avviene in due fasi:

Naming

- Risolto associando staticamente un identificatore (ad es. un numero) all'interfaccia del servizio.

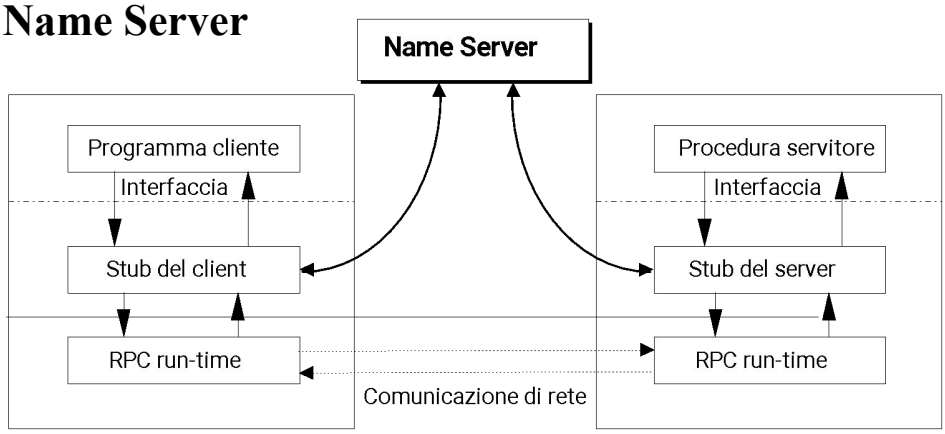
Addressing

In questa fase si cercano gli eventuali servitori pronti per il servizio (utilizzo di un *sistema di nomi*). Possibilità:

1. Client può chiedere direttamente indirizzo al server usando broadcast/multicast
2. Client può interrogare un ***name server***, che registra tutti i servitori

Di solito il binding avviene meno frequentemente delle chiamate stesse, in genere si usa lo stesso binding per molte richieste e chiamate allo stesso server.

Binding e Name Server



Il **name server** consente il collegamento tra il Client e il Server.

Operazioni di ricerca, registrazione, aggiornamento, eliminazione:

- **lookup** (servizio, versione, &server)
- **register** (servizio, versione, server)
- **unregister** (servizio, versione, server)

Attenzione: se il nome del server dipende dal nodo di residenza, ogni variazione deve essere comunicata al name server.

Naming

Diverse possibili architetture:

Centralizzata
Un server centrale che mantiene la lista di tutti i servizi all'interno della rete

- Single point of failure

Decentralizzata
Un server in ciascun host che mantiene una lista di servizi locali

- Il chiamante deve sapere in anticipo quale host fornisce il servizio desiderato

Ibrida
Un server centrale, possibilmente replicato, che mantiene una lista di fornitori di servizi

Naming in Sun/ONC RPC: il port mapper

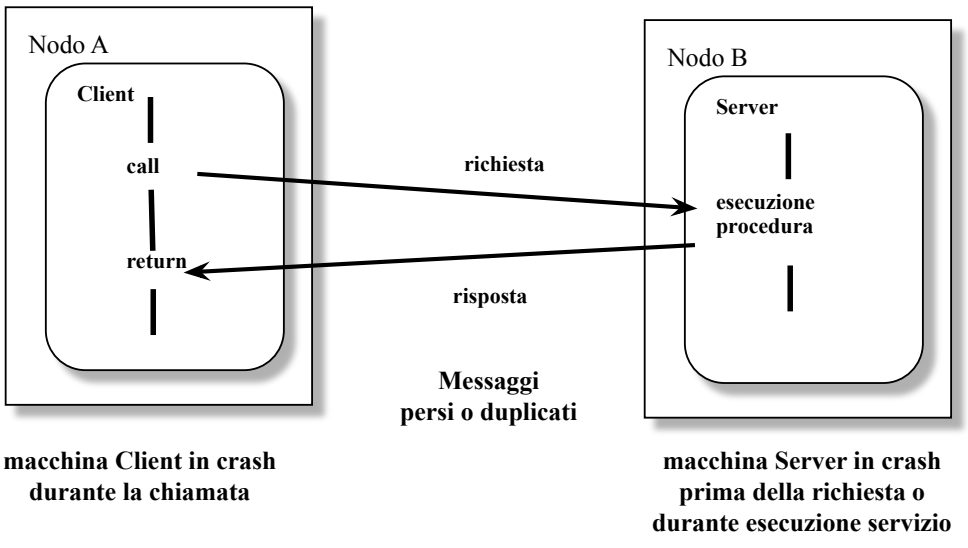
In Sun/ONC RPC il naming è completamente decentralizzato, e viene realizzato tramite il port mapper.

`portmap` è un servizio in ascolto sulla porta 111 che fornisce le funzionalità di lookup e registrazione servizi. Ogni servizio è univocamente identificato da un program ID. I servizi possono avere versioni diverse:

```
$ rpcinfo -p
  program vers proto   port
   100000    2   tcp    111  portmapper
   100000    2   udp    111  portmapper
   100003    2   udp   2049  nfs
   100003    3   udp   2049  nfs
   100003    4   udp   2049  nfs
   100003    2   tcp   2049  nfs
   100003    3   tcp   2049  nfs
   100003    4   tcp   2049  nfs
```


Malfunzionamenti e RPC

Un'importante differenza tra RPC e chiamate di procedure locali riguarda i possibili **malfunzionamenti**



Malfunzionamenti e RPC

Possibili malfunzionamenti:

- la macchina del **Client** può andare in **crash** durante la chiamata
- i **messaggi** possono essere **persi oppure duplicati**, a causa di malfunzionamenti sul canale di comunicazione
- la macchina del **Server** può andare in **crash** prima della richiesta o durante l'esecuzione del servizio prima di mandare la risposta

INOLTRE

- In caso di Server stateful, in seguito a crash il Server potrebbe **perdere** lo **stato** dell'interazione con il Client

- **Orfani**, sono così dette le procedure di servizio correntemente in esecuzione, ma richieste da Client andati in crash. Per servizi molto pesanti sarebbe opportuno notificare il crash dei Client al Server in modo da terminare gli orfani

A seconda delle misure che vengono prese per fronteggiare questi guasti, si ottengono **differenti semantiche** di chiamata di procedura remota.

Semantica may-be

Nel caso in cui **non si prendano misure per la tolleranza ai guasti**, si ottiene una semantica di chiamata di procedura remota di tipo **may-be**.

Se la chiamata fallisce (tipicamente per il termine di un time-out) il Client non può desumere cosa sia successo. Possibili casi:

- perdita messaggio di richiesta
- server crash
- procedura remota eseguita ma perdita messaggio di risposta

Questa semantica è chiamata may-be in quanto in caso di fallimento della chiamata il Client NON è in grado di dire se la procedura remota sia stata eseguita o meno.

Inoltre, se la chiamata ha successo, la procedura remota potrebbe essere stata eseguita anche più di una volta (Client invia solo una richiesta, ma ci possono essere messaggi duplicati dal servizio di comunicazione sottostante).

Di semplice realizzazione ma solleva problemi di consistenza di stato sul Server.

Semantica at-least-once

Nel caso in cui il **Client** stub decida di **riprovare a spedire il messaggio di richiesta** un numero N di volte. Se la chiamata ha successo, la procedura è stata eseguita una o più volte (richieste e messaggi duplicati), da cui at-least-once (almeno una volta).

Se la chiamata fallisce il cliente può pensare a:

- malfunzionamento permanente di rete
- server crash

Questo tipo di semantica può andare bene nel caso di **procedure idempotenti**, cioè che possono essere eseguite molte volte con lo stesso effetto sullo stato di interazione C/S.

Esempio, server stateless di NFS

Semantica at-most-once

Il **Server** potrebbe tenere traccia degli identificatori delle richieste e **scartare richieste duplicate**. Inoltre potrebbe memorizzare il messaggio di risposta e ritrasmetterlo fino al ricevimento di un **acknowledge da parte del Client.**

Questo permette di ottenere una semantica di tipo at-most-once (al massimo una), in quanto è garantito che in ogni caso la procedura remota è:

- non eseguita
- eseguita solo parzialmente (in caso di server crash)
- eseguita una sola volta

Semantica exactly-once

Il Server potrebbe implementare le procedure come **transazioni atomiche**, in modo che le procedure siano eseguite del tutto o per niente.

Questo permette di ottenere una semantica di tipo exactly-once (esattamente una), in quanto è garantito che in ogni caso la procedura remota è:

- non eseguita
- eseguita una sola volta

Questa semantica rappresenta il **caso ideale**, le implementazioni di RPC solitamente presentano semantica di tipo at-most-once.

Malfunzionamenti Client e orfani

In caso di crash del client, si deve trattare il caso degli **orfani**, con diverse strategie:

- **sterminio**, ogni orfano risultato di un crash viene distrutto
- **terminazione a tempo**, ogni calcolo ha una scadenza, oltre la quale è automaticamente abortito
- **reincarnazione** (a epoche), il tempo viene diviso in epoche; tutto quello che è relativo a un'epoca precedente è considerato obsoleto

Eterogeneità e Conversione rappresentazione dati

Eterogeneità Internet (macchine e reti). Il Client e il Server possono eseguire su architetture diverse che usano differenti rappresentazioni dei dati:

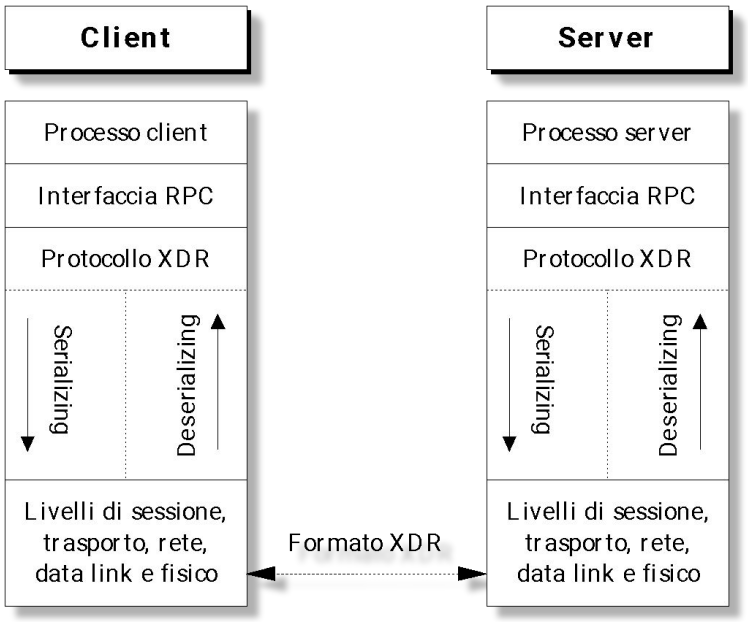
- caratteri (ASCII, ISO8859-*, Unicode UTF-*,…)
- interi (dimensione, complemento a 1 o a 2)
- lunghezza (interi di 2 o 4 byte)
- reali (lunghezza exp e mantissa, formato, ...)
- ordine byte all'interno di una parola (little endian Vs. big endian)

Per comunicare tra nodi eterogenei due soluzioni:

1. ogni nodo converte i dati per il destinatario (prestazioni)
2. concordare un formato comune di rappresentazione dei dati (flessibilità)

Importanza degli standard:
XDR (Sun Microsystems)
ASN.1/X.680 (ITU-T/OSI)
XML (W3C)

eXternal Data Representation (XDR)

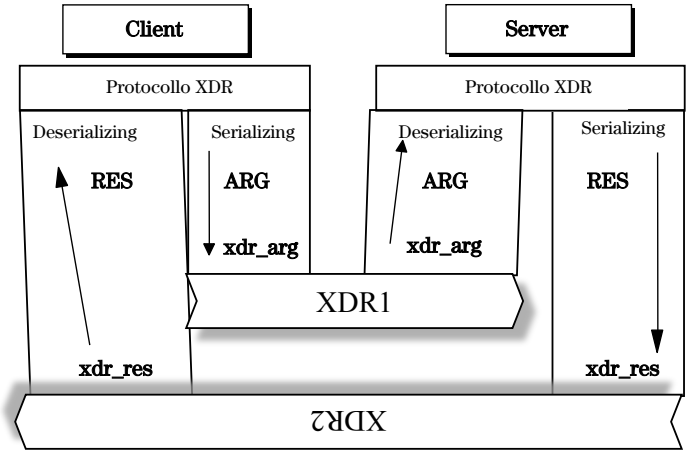


eXternal Data Representation (XDR)

XDR fornisce un insieme di procedure di conversione per trasformare la rappresentazione **nativa** dei dati in una **rappresentazione esterna XDR** e viceversa.

XDR fa uso di uno **stream**, un buffer, che permette di creare un messaggio con i dati in forma XDR.

I dati vengono inseriti/estratti nello/dallo stream XDR uno alla volta, operazioni di **serializzazione**/**deserializzazione.**



Flussi XDR (stream)

eXternal Data Representation (XDR)

Esempio (serializzazione):

```
…
XDR *xdrs;
char buf[BUFSIZE];
…
xdrmem_create(xdrs, buf, BUFSIZE, XDR_ENCODE);
…
i=260;
xdr_int(xdrs, &i);
```

Creazione stream XDR

Inserimento nello stream di un intero, convertito in formato XDR

La **deserializzazione** avviene nello stesso modo, con le **stesse routine**, usate in verso opposto, si specifica flag XDR_ DECODE in xdrmem_create().

eXternal Data Representation (XDR)

Funzioni built-in di conversione

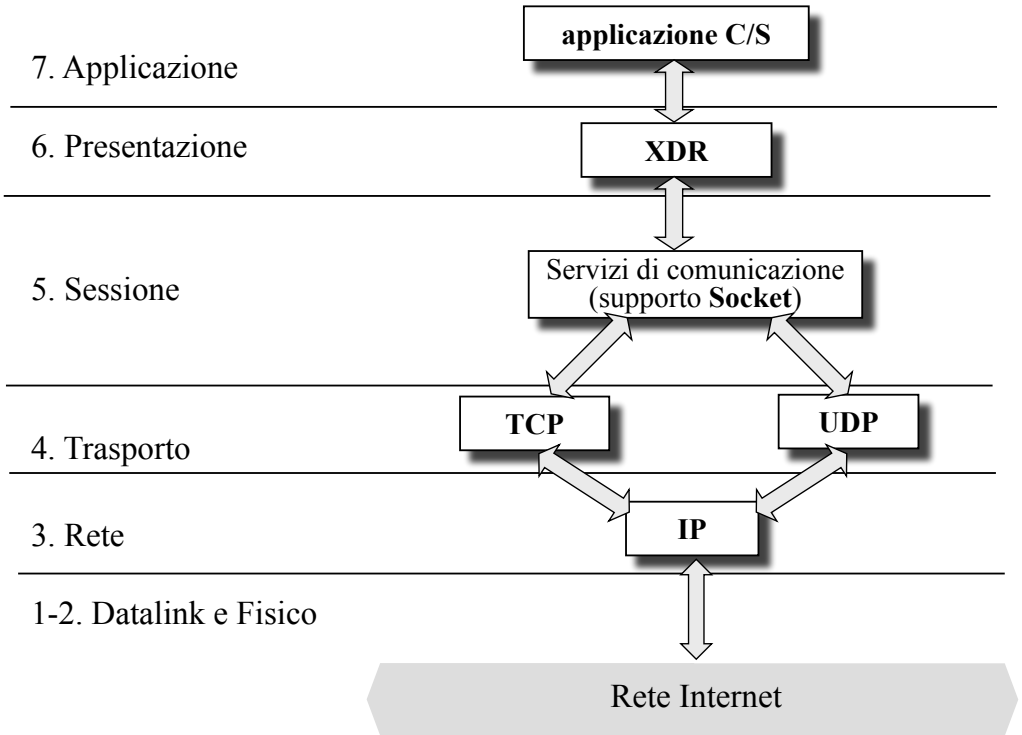
Funzione built-in	Tipo di dato convertito
xdr_bool()	Logico
xdr_char() *xdr_u_char()*	Carattere
xdr_short() *xdr_u_short()*	Intero a 16 bit
xdr_enum()	Enumerazione
xdr_float()	Virgola mobile
xdr_int() *xdr_u_int*	Intero
xdr_long() *xdr_u_long()*	Intero a 32 bit
xdr_void()	Nullo
xdr_opaque()	Opaco (byte senza significato particolare)
xdr_double()	Doppia precisione

Funzioni per tipi composti

Funzione	Tipo di dato convertito
xdr_array()	Vettori con elementi di tipo qualsiasi
xdr_vector()	Vettori a lunghezza fissa
xdr_string()	Sequenza di caratteri con NULL
xdr_bytes()	Vettore di bytes senza terminatore
xdr_reference()	Riferimento ad un dato
xdr_pointer()	Riferimento ad un dato, incluso NULL
xdr_union()	Unioni

Attenzione: nel caso di informazioni strutturate, XDR ne esegue la serializzazione con **pointer chasing** e **flattening.**

RPC e livelli OSI



Esempio di codice in Sun ONC RPC

Vediamo una semplice implementazione di remote file listing (ls).

rls.x: interfaccia IDL del servizio – parte 1/2

```
/* definisci massima dimensione stringhe */
const MAXNAMELEN = 255;

/* parametro: nome directory */
typedef string nametype<MAXNAMELEN>;

/* valore di ritorno: lista di file */
typedef struct namenode *namelist;
struct namenode {
   nametype name;  /* nome del file */
   namelist pNext; /* prossimo file */
};
```


Esempio di codice in Sun ONC RPC

rls.x: interfaccia IDL del servizio – parte 2/2

```
/* risultato della chiamata */
union readdir_res switch (int error_code) {
case 0:
   namelist list;   /* ok */
default:
   void;     /* errore */
};

/* definizione dell'interfaccia */
program DIRPROG {
   version DIRVERS {
      readdir_res READDIR(nametype) = 1;
   } = 1; /* identificativo interfaccia */
} = 0x20000000; /* identificativo servizio */
```

Esempio di codice in Sun ONC RPC

Generazione di stub e skeleton:

```
mauro@orion:code/rpc_example$ rpcgen -a -C rls.x
```

Il comando **`rpcgen`** genera diversi file:

Makefile.rls – Makefile (da customizzare)
rls_client.c – Template per il client (da customizzare)
rls_clnt.c – Stub (codice stub per il client)
rls.h – Interfaccia
rls_server.c – Template per il server (da customizzare)
rls_svc.c – Skeleton (codice stub per il server)
rls_xdr.c – Funzioni XDR

Esempio di codice in Sun ONC RPC (codice server 1/2)

```
readdir_res *
readdir_1_svc(nametype *dirname, struct svc_req *rqstp)
{
  namelist nl;
  namelist *nlp;
  struct direct *d;
  DIR *dirp = NULL;
  static readdir_res res; /* must be static */

  /* open the directory */
  dirp = opendir(*dirname);
  if (dirp == NULL) {
      res.error_code = errno;
      return &res;
  }
```

Esempio di codice in Sun ONC RPC (codice server 2/2)

```
  /* free previously allocated memory */
  xdr_free((xdrproc_t)xdr_readdir_res, (char*)&res);

  /* read directory */
  nlp = &res.readdir_res_u.list;
  while (d = readdir(dirp)) {
      nl = *nlp = (namenode *)malloc(sizeof(namenode));
      nl->name = strdup(d->d_name);
      nlp = &nl->pNext;
  }

  *nlp = NULL;
  res.error_code = 0;
  closedir(dirp);
  return &res;
}
```


Esempio di codice in Sun ONC RPC (codice client)

```
/* create handle for service call */
clnt = clnt_create(host, DIRPROG, DIRVERS, "udp");
if (clnt == NULL) {
    clnt_pcreateerror(host); exit(1);
}

result = readdir_1(&dirname, clnt); /* call remote procedure */
if (result == (readdir_res *)NULL)
    clnt_perror(clnt, "call failed");
if (result->error_code != 0) {/* check for errors */
    errno = result->error_code;
    perror(dirname); exit(1);
}
/* print results */
for (nl = result->readdir_res_u.list;
     nl != NULL; nl = nl->pNext) {
         printf("%s\n", nl->name);
}

clnt_destroy(clnt); /* destroy client handle */
```

gRPC (Google Remote Procedure Call)

Modello RPC sviluppato da Google, pensato per essere open e cross-language.

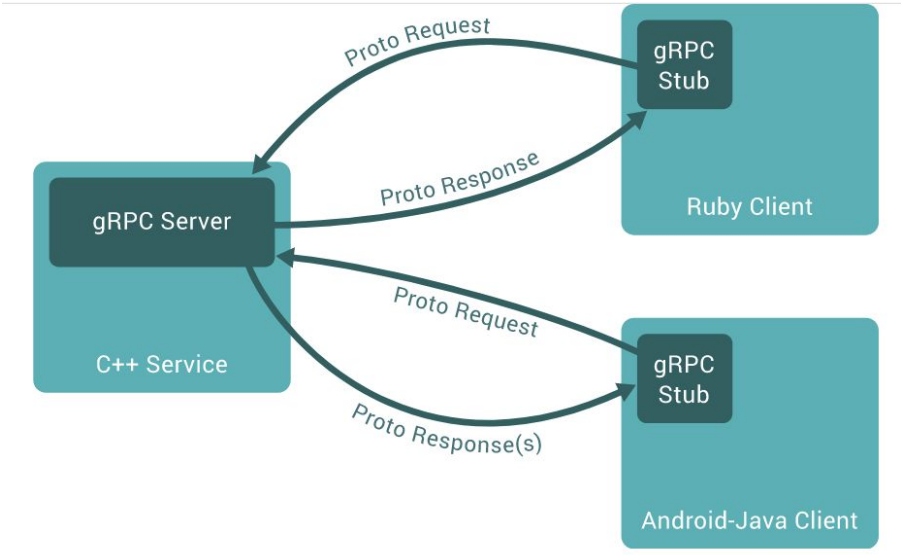
gRPC utilizza i Google Protocol Buffer (Protobuf) per la definizione dell'interfaccia del servizio e dei messaggi scambiati tra stub (chiamato anche client) e server.

L'implementazione è a carico del programmatore in un linguaggio di programmazione a scelta fra quelli supportati: (C++, Java, Ruby, etc…) https://grpc.io/docs/quickstart/

Come in SUN ONC/RPC si parte dalla definizione di servizio che include metodi offerti (rpc) e definizione dei tipi di dato scambiati tra client e server.

gRPC è stato pensato per essere cross-language, lasciando totale libertà di implementazione al programmatore. Il server e il Client possono essere implementati in un linguaggio di programmazione differenti (uno dei punti forti di gRPC).

gRPC (Google Remote Procedure Call)



https://grpc.io/docs/guides/

gRPC Definizione del servizio

Come per il precedente esempio vogliamo implementare una semplice applicazione di remote file listing (ls).

Un servio gRPC viene definito *specificandone la sua interfaccia* utilizzando un Google Protocol Buffer. Il nostro **servizio** sarà quindi definito in un file .proto.

```
option java_package = "it.unife.reti.grpc.example"

service RemoteLs  {
    rpc Ls(dirname) returns (FileList) {}
}

message DirName {
    string name = 1;
}

message FileList {
    repeated string file = 1;
}
```


gRPC Generazione stub e server (1/2)

In questo esempio, implementeremo il servizio di **RemoteLs** (sia Server che Client) in Java.

Come per un classico RPC, l'interfaccia del servizio deve essere compilata per la generazione del codice dello stub e del server. La compilazione dell'interfaccia richiede l'utilizzo del protobuf compiler e il gRPC plugin per Java.

In progetti più complessi è **caldamente consigliato** utilizzare strumenti per lo sviluppo del software come Gradle, che aiutino ad automatizzare il processo di compilazione sia del codice Java che del'interfaccia gRPC.

La gestione di librerie e dipendenze esterne è complessa, strumenti come Gradle tendono in genere a semplificare e ottimizzare il lavoro.

gRPC Generazione stub e server (2/2)

Per compilare il gRPC-Java plugin possiamo seguire le istruzioni al link fornito. La compilazione produce un eseguibile che verrà utilizzato dal protobuf compiler per compilare il nostro servizio gRPC.

Per compilare il servizio gRPC dobbiamo compilare il Protocol Buffer nel seguente modo:

Per la generazione del codice del servizio:

```
protoc --plugin=./protoc-gen-grpc-java --grpc-java_out="."
--proto_path="./" remote_ls.proto
```

Per la generazione dei tipi di dato (DirName, FileList):

```
protoc --java_out=. --proto_path=. remote_ls.proto
```

Se la compilazione ha avuto successo possiamo trovare i file generati dalla compilazione all'interno del nostro package definito nel file **.proto**

gRPC Implementazione del Server (1/4)

Per implementare il Server del nostro servizio dobbiamo creare una nuova classe Java che:

- implementi le funzioni definite nell'interfaccia (remote ls)
- istanzi e mandi in esecuzione il servizio

L'implementazione del servizio è definita in una classe statica chiamata **RemoteLsImplementation** che **estende** la classe astratta creata dalla compilazione del servizio.

```
public class RemoteLsServer {
    //implementazione del servizio
    private Server server = null; private int port;
        static class RemoteLsImplementation extends
        RemoteLsGrpc.RemoteLsImplBase {
            @Override
            public void ls(…)
        }
}
```

gRPC Implementazione del Server (2/4)

Il metodo **ls** deve avere la stessa signature definita nella classe astratta:
• DirName contiene la stringa rappresentante il nome della directory
• StreamObsever viene utilizzato da gRPC per gestire il flusso di esecuzione della chiamata **rpc**

```
public void ls(DirName dirname, StreamObserver<FileList>
responseObserver) {
        File dir = new File(dirname.getName());
        FileList.Builder flb = FileList.newBuilder();
        for(String file: dir.list()) {
            flb.addFile(file);
        }
        // flb.build() restituisce un oggetto di tipo
        // FileList
        responseObserver.onNext(flb.build());
        responseObserver.onCompleted();
}
```


gRPC Implementazione del Server (3/4)

Il servizio va poi messo in esecuzione istanziando ServerBuilder nel costruttore della nostra calasse Server:

```
public RemoteLsServer(int port) {
    this.port = port;
        ServerBuilder sb = ServerBuilder.forPort(port)
        .addService(new RemoteLsImplementation()); server =
        sb.build();
}
```

ServerBuilder viene inizializzato passandogli una istanza dell'implementazione del servizio, in questo caso RemoteLsImplementation.

gRPC Implementazione del Server (4/4)

Il server deve essere poi mandato in esecuzione, chiamandone il metodo start all'interno del metodo main della nostra classe Server.

```
public static void main(String[] args) throws IOException,
            InterruptedException {
    final RemoteLsServer rlsServer = new RemoteLsServer(8081);
rlsServer.start();     rlsServer.blockUntilShutDown();
}
```


gRPC Implementazione del Client (1/2)

Per implementare il Client del nostro servizio dobbiamo creare una nuova classe Java che crei un'istanza dello stub (in questo caso bloccante) definito dal processo di compilazione dell'interfaccia:

```
public class RemoteLsClient {
    private final ManagedChannel channel;
    private final RemoteLsBlockingStub blockingStub;
    ….

    public RemoteLsClient(String host, int port) {
        channel = ManagedChannelBuilder
                .forAddress(host,port
                    .usePlaintext().build()
        blockingStub = RemoteLsGrpc
                    .newBlockingStub(channel);

    }
```

gRPC Implementazione del Client (2/2)

Il client poi potrà chiamare il metodo del nostro servizio utilizzando lo stub:

```
public void ls(String directory) {
    DirName dir = DirName
        .newBuilder().setName(directory).build();
FileList filelist = blockingStub.ls(dir);
    for (int i = 0; i < filelist.getFileCount(); i++) {
    System.out.println("" + filelist.getFile(i));        }
}


 public static void main (String[] args) {
    String dirname = args[0];
    RemoteLsClient client = new RemoteLsClient("127.0.0.1", 8081);
    client.ls(dirname);
}
```


gRPC Compilazione Client e Server

Per compilare il nostro servizio di remote listing abbiamo bisogno di “qualche” libreria esterna; dato che non utilizzeremo Gradle, dobbiamo aggiungere queste librerie al classpath del Java Compiler.

Nell’esempio riportato le dipendenze sono all’interno della directory dependencies.

```
javac -cp dependencies/\* it/unife/reti/grpc/example/*.java -d
classes/
```

gRPC Esecuzione Client e Server

Nello stesso modo, dobbiamo aggiungere le dipendenze esterne e i compilati al Java classpath per poter eseguire il codice del client e server gRPC:

```
java -cp .:dependencies/*:classes/
it.unife.reti.grpc.example.RemoteLsServer
```

```
java -cp .:dependencies/*:classes/
it.unife.reti.grpc.example.RemoteLsClient <dirname>
```