Università degli Studi di Roma "Tor Vergata"

Facoltà di Ingegneria

Comunicazione nei Sistemi Distribuiti

Corso di Sistemi Distribuiti

Valeria Cardellini

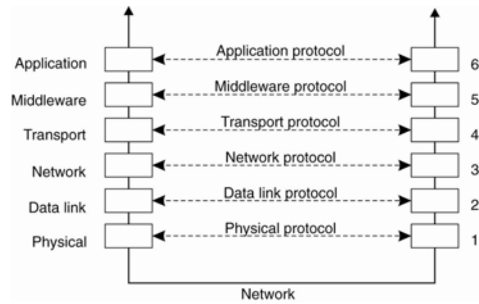
Anno accademico 2008/09

Comunicazione nei SD

- Si basa sempre sullo scambio di messaggi
  - Invio e ricezione di messaggi (a basso livello)
- Per permettere lo scambio di messaggi, le parti devono accordarsi su diversi aspetti
  - Quanti volt per segnalare un bit 0 e quanti per un bit 1?
  - Come fa il destinatario a sapere quale è l'ultimo bit del messaggio?
  - Quanto è lungo un intero?
  - …
- Soluzione: suddividere il problema in livelli
  - Ciascun livello in un sistema comunica con lo stesso livello nell'altro sistema
  - Modello di riferimento ISO/OSI

Protocolli a livelli

- Adattamento del modello di riferimento tradizionale per le comunicazioni di rete
  - Protocolli di livello più basso
  - Protocollo di trasporto
  - Protocollo middleware
  - Protocollo applicativo



- Livello **middleware**
  - Fornisce servizi comuni e protocolli general-purpose, di alto livello, indipendenti dalle applicazioni, che possono essere usati da altre applicazioni diverse
  - Ampio insieme di protocolli di comunicazione
  - (Un)marshaling dei dati, necessario per sistemi integrati
  - Protocolli di naming, in modo che applicazioni diverse possano condividere risorse
  - Protocolli di sicurezza, per consentire ad applicazioni diverse di comunicare in modo sicuro
  - Meccanismi di scaling, come il supporto per la replicazione ed il caching

Tipi di comunicazione

- Distinguiamo la comunicazione in base a:
  - Persistenza
    - Comunicazione persistente o transiente
  - Sincronizzazione
    - Comunicazione sincrona o asincrona
  - Dipendenza dal tempo
    - Comunicazione discreta e a stream

Tipi di comunicazione (2)

- Comunicazione persistente
  - Il messaggio immesso viene memorizzato dal middleware per tutto il tempo necessario alla consegna
  - Non occorre che il processo mittente continui l'esecuzione dopo l'invio del messaggio
  - Non è necessario che il processo destinatario sia in esecuzione quando il messaggio è inviato
- Comunicazione transiente
  - Il messaggio è memorizzato dal middleware solo finché i processi mittente e destinatario sono in esecuzione
  - Se la consegna non è possibile, il messaggio viene cancellato
  - E' il caso della comunicazione di rete a livello di trasporto: i router memorizzano ed inoltrano; se non è possibile inoltrare, cancellano

Tipi di comunicazione (3)

- Comunicazione sincrona
  - Una volta sottomesso il messaggio, il processo mittente si blocca finché l'operazione non è completata
  - L'invio e la ricezione sono operazioni bloccanti
  - Fino a quando il mittente si blocca?
    - Finché il middleware non prende il controllo della trasmissione
    - Finché il messaggio non viene consegnato al destinatario
    - Finché il messaggio non viene elaborato dal destinatario
- Comunicazione asincrona
  - Una volta sottomesso il messaggio, il processo mittente continua l'elaborazione: il messaggio viene memorizzato temporaneamente dal middleware fino ad avvenuta trasmissione
  - L'invio è un'operazione non bloccante
  - La ricezione può essere bloccante o non bloccante

Tipi di comunicazione (4)

- Comunicazione discreta
  - Ogni messaggio costituisce un'unità di informazione completa
- Comunicazione a stream (streaming)
  - Comporta l'invio di molti messaggi, in relazione temporale tra loro o in base all'ordine di invio, che servono a ricostruire un'informazione completa

Tipi di comunicazione e loro combinazione

- Quali sono le possibili combinazioni reali tra persistenza e sincronizzazione?
- Comunicazione persistente e asincrona
  - Ad es., posta elettronica
- Comunicazione persistente e sincrona
  - Mittente bloccato fino alla copia (garantita) del messaggio presso il destinatario
- Comunicazione transitoria e asincrona
  - Mittente non attende ma messaggio perso se destinatario non raggiungibile (ad es. UDP)
- Comunicazione transitoria e sincrona
  a) mittente bloccato fino alla copia del messaggio nel destinatario
  b) mittente bloccato fino alla copia (non garantita) del messaggio nello spazio del destinatario (RPC asincrona)
  c) mittente bloccato fino alla ricezione di un messaggio di risposta dal destinatario (RPC sincrona e RMI)

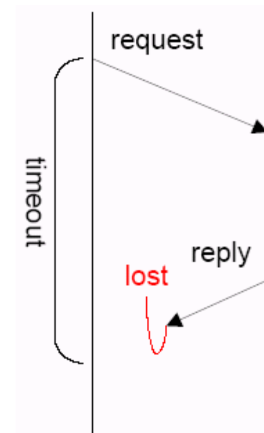
Semantica della comunicazione

- Il messaggio di richiesta e/o risposta può andare perduto
  - La comunicazione su rete è sempre soggetta a fallimento
- Quale è la semantica di comunicazione in presenza di errori?
- Dipende dalla combinazione di tre meccanismi di base:

1. Lato client: riprova (Request Retry – RR1)
   - Il client continua a provare fino a ottenere risposta o la certezza del guasto del destinatario
2. Lato server: filtra i duplicati (Duplicate Filtering – DF)
   - Il server scarta gli eventuali duplicati di richieste provenienti dallo stesso client
3. Lato server: ritrasmetti le risposte (Result Retransmit – RR2)
   - Il server conserva le risposte per poterle ritrasmettere senza ricalcolarle
   - Fondamentale ove l'operazione non fosse idempotente
     - Operazione idempotente: la sua esecuzione ripetuta produce gli stessi effetti che se l'operazione fosse eseguita una sola volta

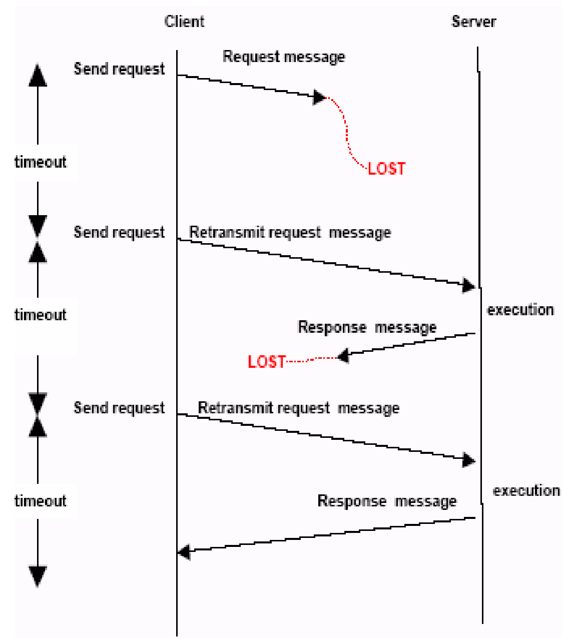
Semantica della comunicazione (2)

- Semantica may-be (forse)
  - Il messaggio può arrivare o meno
  - Nessun meccanismo in uso
  - Non si fanno azioni per garantire affidabilità
  - Ad es. best-effort in IP o UDP



Semantica della comunicazione (3)

- Semantica at-least-once (almeno una volta)
  - Il messaggio, se arriva, arriva almeno una volta
  - In caso di insuccesso nessuna informazione
  - Il client usa RR1 ma il server non usa né DF né RR2
    - Il server non si accorge delle ritrasmissioni
  - All'arrivo di una risposta, il client non sa quante volte sia stata calcolata dal server: quindi, non conosce per certo lo stato del server



Semantica della comunicazione (4)

- Semantica at-most-once (al più una volta)
  - Il messaggio, se arriva, arriva al più una volta
    - Il client sa che la risposta è stata calcolata una sola volta
    - La risposta non arriva solamente in presenza di guasti permanenti del server
  - Tutti i meccanismi in uso
    - Il client effettua ritrasmissioni
    - Il server mantiene uno stato per riconoscere i messaggi già ricevuti e per non eseguire operazioni più di una volta
  - Adatta per qualunque operazione
    - Anche non idempotente
  - Semantica che non mette vincoli sulle azioni conseguenti
    - Manca un coordinamento tra client e server
  - Può produrre inconsistenza sull'accordo tra mittente e ricevente

Semantica della comunicazione (5)

- Semantica exactly-once (esattamente una volta)
  - Il messaggio arriva una volta sola oppure entrambi conoscono lo stato finale dell'altro
  - Accordo completo sull'interazione
    - Il messaggio non è arrivato o non è stato considerato da entrambi
  - Semantica con conoscenza concorde dello stato dell'altro e senza ipotesi di durata massima del protocollo
  - Ha bisogno di ulteriori meccanismi (ad es. replicazione trasparente) per tollerare guasti di lato server

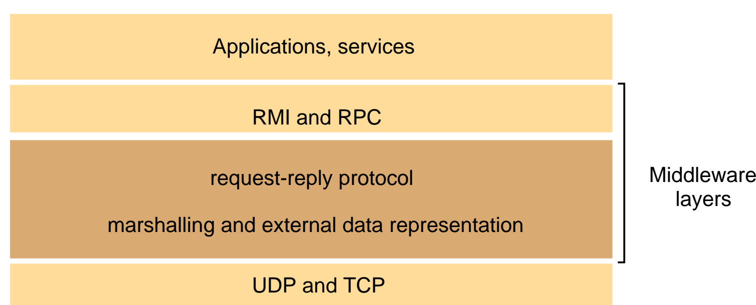
Programmazione di applicazioni di rete

- Programmazione di rete *esplicita*
  - Chiamata diretta all'API socket e scambio esplicito di messaggi
  - Usata nella maggior parte delle applicazioni di rete note (ad es. Web browser, Web server)
  - La distribuzione non è trasparente
  - Gran parte del peso dello sviluppo sulle spalle del programmatore

- Come innalzare il livello di astrazione della programmazione distribuita? Fornendo uno strato intermedio fra SO ed applicazioni (middleware) che:
  - Nasconda la complessità degli strati sottostanti
  - Liberi il programmatore da compiti automatizzabili
  - Migliori la qualità del software mediante il riuso di soluzioni consolidate, corrette ed efficienti

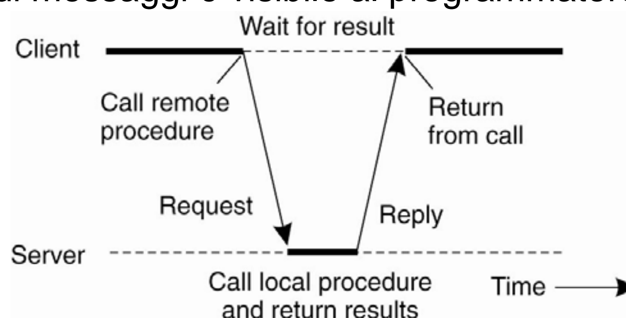
Programmazione di applicazioni di rete (2)

- Programmazione di rete *implicita*
  - **Chiamata di procedura remota (RPC)**
    - L'applicazione distribuita è realizzata usando le chiamate di procedura, ma il processo chiamante (client) ed il processo contenente la procedura chiamata (server) possono essere su macchine diverse
  - **Invocazione di metodo remoto (RMI)**
    - L'applicazione distribuita è realizzata invocando i metodi di un oggetto di una applicazione Java in esecuzione su una macchina remota
  - Trasparenza della distribuzione



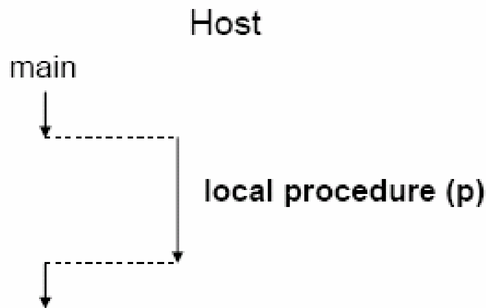
Chiamate a procedura remota

- Remote Procedure Call (RPC)
- Idea (proposta da Birrel e Nelson, 1984): utilizzare il modello client server con la stessa semantica di una chiamata di procedura
  - Un processo sulla macchina A invoca una procedura sulla macchina B
  - Il processo chiamante su A viene sospeso
  - Ha luogo l'esecuzione della procedura chiamata su B
  - Input ed output sono veicolati da parametri
  - Nessun passaggio di messaggi è visibile al programmatore

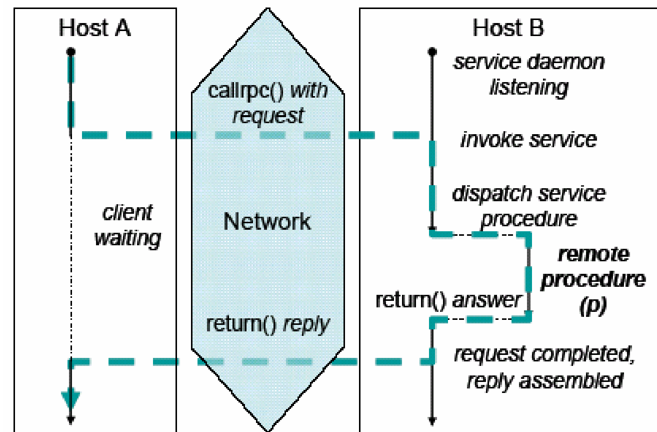


Chiamata di una procedura

- Procedura p locale



- Procedura p remota

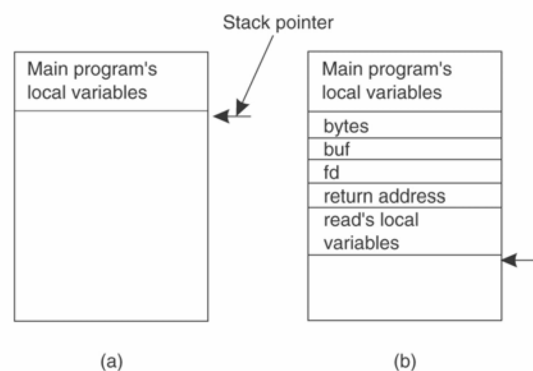


Chiamata di una procedura “locale”

- Esempio di chiamata “locale” (convenzionale):

  count = read(fd, buf, nbytes)

- La procedura chiamante inserisce sullo stack i parametri e l’indirizzo di ritorno
- La procedura chiamata (read) mette sullo stack il valore di ritorno e restituisce il valore alla procedura chiamante



Tipi di passaggio dei parametri

- Passaggio per valore (call by value)
  - I dati sono copiati nello stack
  - Il chiamato agisce su tali dati e le modifiche non influenzano il chiamante
- Passaggio per riferimento (call by reference)
  - L'indirizzo (puntatore) dei dati è copiato nello stack
  - Il chiamato agisce direttamente sui dati del chiamante
- Passaggio per copia/ripristino (call by copy/restore)
  - La variabile viene copiata nello stack dal chiamante e ricopiata dopo la chiamata sovrascrivendo il valore originale del chiamante
  - Raramente implementato nei linguaggi

Problemi RPC

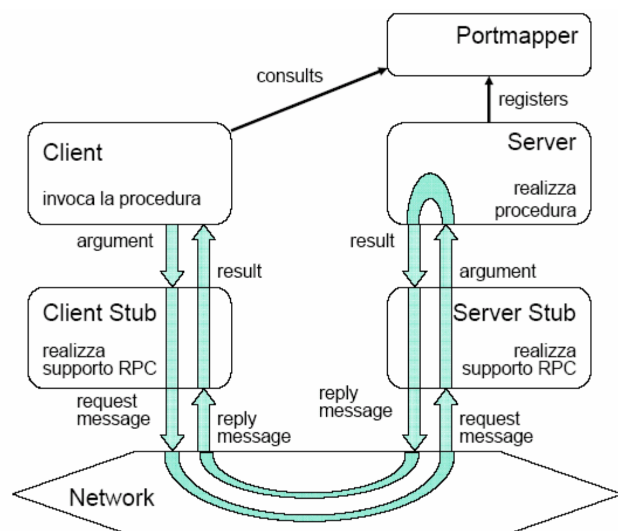
- Quali sono i problemi da risolvere per riuscire a realizzare la semantica della chiamata di procedura in modo trasparente e senza sapere dove si trova la procedura chiamata?
- Diversa rappresentazione dei dati sulle due macchine
- Procedura chiamante e chiamata in esecuzione su macchine diverse con un diverso spazio di indirizzi: come realizzare il passaggio dei parametri?
- In presenza di malfunzionamenti, cosa può la procedura chiamante considerare certo rispetto all'esecuzione della procedura chiamata?
  - Nel caso di procedura locale: semantica exactly-once
  - Nel caso di procedura remota: semantica at-least-once oppure at-most-once
- Come localizzare su quale macchina viene eseguita la procedura chiamata?

Requisiti per implementazione RPC

- Il supporto scambia messaggi per consentire
  - Identificazione dei messaggi di chiamata e risposta
  - Identificazione univoca della procedura remota
- Il supporto gestisce l'eterogeneità dei dati scambiati
  - Marshaling/unmarshaling dei parametri
  - Serializzazione dei parametri
- Il supporto gestisce alcuni errori dovuti alla distribuzione
  - Implementazione errata
  - Errori dell'utente
  - Roll-over (ritorno indietro)

Architettura RPC

- Modello con **stub** (adattatore)
- Il cliente invoca uno stub (client stub), che si incarica di tutto:
  - dal recupero del server, al trattamento dei parametri e dalla richiesta al supporto run-time, al trasporto della richiesta
- Il server riceve la richiesta dallo stub relativo (server stub), che si incarica del trattamento dei parametri dopo avere ricevuto la richiesta pervenuta dal trasporto. Al completamento del servizio, lo stub rimanda il risultato al client
- Lo sviluppo prevede la massima trasparenza
  - Stub prodotti in modo automatico
  - Lo sviluppatore progetta e si occupa solo delle reali parti applicative e logiche

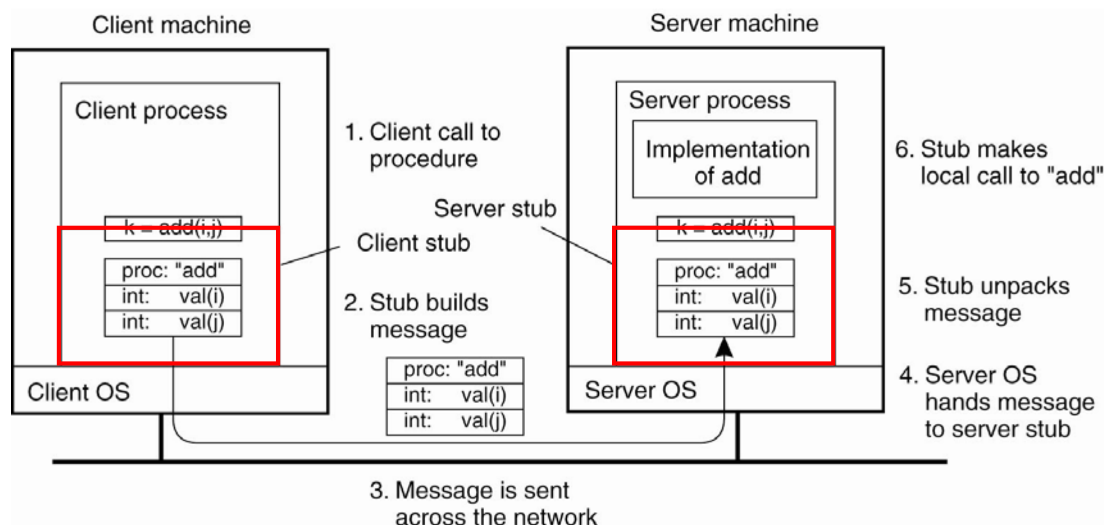


Passi di una RPC

1. La procedura chiamata, lato client, richiama il client stub usando una normale chiamata di procedura locale
2. Il client stub costruisce un messaggio e richiama il SO locale
   - Marshaling dei parametri: operazione di impacchettare i parametri in un messaggio
3. Il SO del client invia il messaggio al SO remoto
4. Il SO remoto passa il messaggio al server stub
5. Il server stub spacchetta il messaggio estraendo i parametri e richiama il server
   - Unmarshaling dei parametri
6. Il server esegue il lavoro e restituisce il risultato al server stub
7. Il server stub lo impacchetta in un messaggio e richiama il suo SO
8. Il SO del server invia il messaggio al SO del client
9. Il SO del client passa il messaggio al client stub
10. Il client stub spacchetta il messaggio e restituisce il risultato al client

Passi di una RPC (2)

- Esempio con passaggio di parametri per valore
  add(i, j)



Problemi per la rappresentazione dei dati

- Client e server potrebbero usare una diversa codifica (encoding) dei dati
  - Codifica di caratteri
  - Rappresentazione di numeri interi e in virgola mobile
  - Ordinamento dei byte (little endian vs big endian)
  - Non solo per tipi di dato elementari ma anche strutturati
- Occorre che i messaggi siano interpretati in modo non ambiguo: possibili soluzioni
  - Inserimento nel messaggio di informazioni sul formato di rappresentazione usato
  - Imposizione di forme canoniche concordate
- Client e server devono coordinarsi anche sul protocollo di trasporto usato per lo scambio di messaggi

Passaggio di parametri per riferimento

- Un riferimento è un indirizzo in memoria
  - E' valido solo nel contesto in cui è usato
- Soluzione: si usa il meccanismo di copia-ripristino
- Il client stub copia i dati puntati nel messaggio e lo invia al server stub
- Il server stub effettua le operazioni con la copia, usando lo spazio di indirizzi della macchina ricevente
- Se occorre effettuare una modifica sulla copia, questa sarà poi riportata dal client stub sul dato originale
- Ottimizzazioni:
  - Se il client stub sa che il riferimento è un parametro di input, non serve che venga ricopiato il suo valore restituito
  - Se si tratta di un parametro di output, non serve effettuare la copia in partenza

Interface Definition Language

- Linguaggio per la definizione delle interfacce (Interface Definition Language - IDL)
- Permette la descrizione delle operazioni remote, la specifica del servizio (detta *firma*) e la generazione degli stub
- Deve consentire l’identificazione non ambigua
  - Identificazione della procedura
    - Uso di nome astratto del servizio, spesso prevedendo versioni diverse del servizio
  - Definizione astratta dei dati da trasmettere in input ed output
    - Uso di un linguaggio astratto di definizione dei dati (uso di interfacce, con operazioni e parametri)
- Supporta lo sviluppo dell’applicazione
  - permettendo di generare automaticamente gli stub dalla interfaccia specificata dall'utente

Binding del server

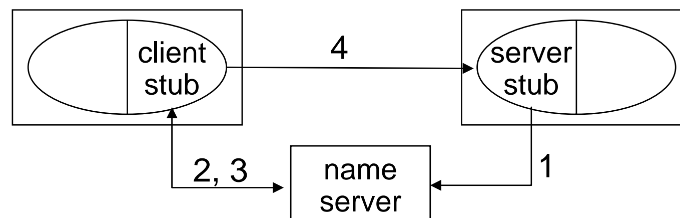
- Il binding stabilisce come ottenere l’aggancio corretto tra il client ed il server in grado di fornire l’operazione
- Due tipologie di binding: statico o dinamico
- Binding statico
  - L’indirizzo del server è cablato all’interno del client
  - Semplice, costo limitato ma mancanza di trasparenza e flessibilità
- Binding dinamico
  - Ritarda la decisione al momento della necessità
  - Costi maggiori, ma trasparenza e flessibilità
    - Ad es. consente di dirigere le richieste sul server più scarico
  - Il costo deve essere comunque limitato ed accettabile

Binding dinamico

- Si distinguono due fasi nella relazione client/server
- **Naming**: fase statica, prima dell'esecuzione
  - Il client specifica a chi vuole essere connesso, con un nome unico identificativo del servizio
  - Si associano dei nomi unici di sistema alle operazioni o alle interfacce astratte e si attua il binding con l'interfaccia specifica di servizio
- **Addressing**: fase dinamica, durante l'esecuzione
  - Il client deve essere realmente collegato al server che fornisce il servizio al momento dell'invocazione
  - Si cercano gli eventuali server pronti per il servizio
  - Addressing esplicito o implicito
  - *Addressing esplicito*: multicast o broadcast dal parte del client attendendo solo la prima risposta
    - Il supporto runtime delle RPC di ogni macchina risponde se il servizio richiesto è fornito da un suo server in esecuzione

Binding dinamico (2)

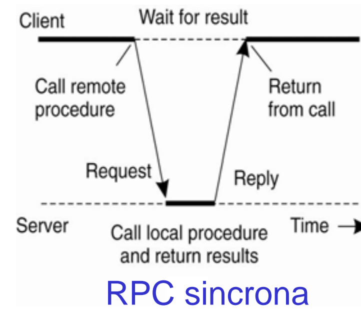
- *Addressing implicito*: uso di un name server (o binder o directory service) che registra tutti i server e agisce su opportune tabelle di binding, prevedendo funzioni di ricerca di nomi, registrazione, aggiornamento, eliminazione



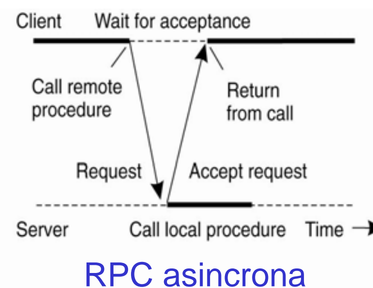
- Frequenza del binding dinamico
  - Ogni chiamata richiede un collegamento dinamico
  - Spesso per questioni di costo dopo un primo legame si usa lo stesso binding ottenuto come se fosse statico
  - Il binding può avvenire meno frequentemente delle chiamate stesse
    - In genere, si usa lo stesso binding per molte chiamate allo stesso server

RPC asincrona

- In modalità sincrona, una chiamata RPC provoca il blocco del client
  - Se il server non deve restituire nessun risultato, il blocco del client non è necessario
- Alcuni sistemi RPC forniscono la possibilità di effettuare RPC asincrone
  - Il client può dedicarsi ad altri task, una volta effettuata la chiamata e ricevuto dal server un acknowledge che la chiamata di procedura è stata ricevuta ed avviata



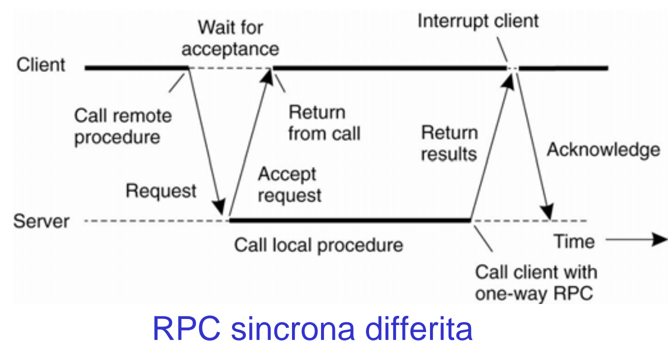
RPC sincrona



RPC asincrona

RPC asincrona (2)

- Se la RPC produce un risultato, si può spezzare l'operazione in due (RPC sincrona differita):
  - Una prima RPC asincrona dal client al server per avviare l'operazione
  - Una seconda RPC asincrona dal server al client per restituire il risultato
- Se il client riprende l'esecuzione senza aspettare l'acknowledge, la RPC asincrona è detta "a senso unico"

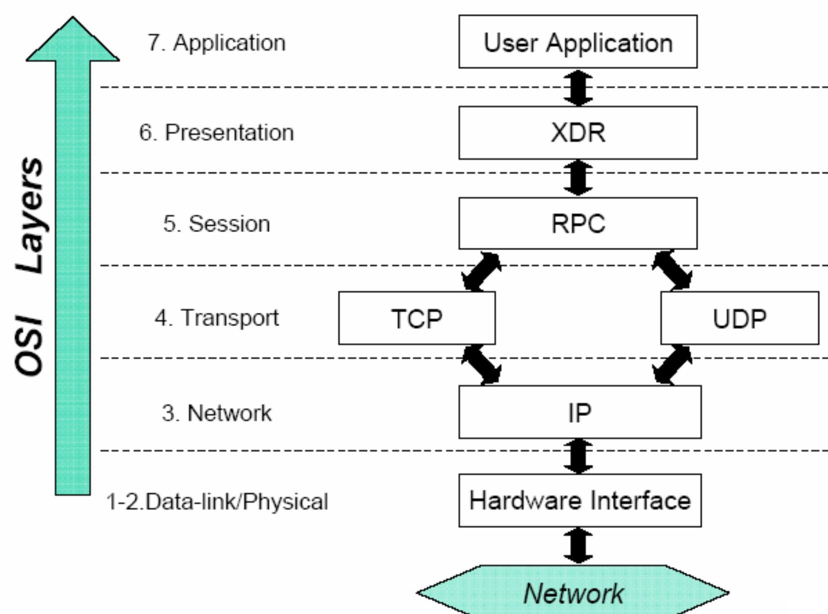


RPC sincrona differita

Implementazione del supporto RPC

- Analizziamo l'implementazione ed il supporto RPC di Sun Microsystems: Open Network Computing (ONC), anche noto come SUN RPC
- ONC è una suite di prodotti che include:
  - eXternal Data Representation (XDR): rappresentazione e conversione dati
  - Remote Procedure Call GENerator (RPCGEN): generazione del client e server stub
  - Portmapper: risoluzione indirizzo del server
  - Network File System (NFS): file system distribuito di Sun
- Molte altre implementazioni, tra cui Distributed Computing Environment (DCE) RPC

SUN RPC e stack OSI



Definizione del programma RPC

- Due parti descrittive in linguaggio RPC

1. Definizioni di programmi RPC: specifiche del protocollo RPC per le procedure (servizi) offerte, ovvero l'identificazione delle procedure ed il tipo di parametri
2. Definizioni XDR: definizioni dei tipi di dati dei parametri
   - Presenti solo se il tipo di dato non è un noto in XDR

- Raggruppate in un unico file con estensione .x
  - Esempio: square.x

Definizione della procedura remota

```
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 */
```

Esempio: square.x

Definizione della procedura remota SQUAREPROC; notare che:

– Ogni definizione di procedura ha un solo parametro d'ingresso e un solo parametro d'uscita

– Gli identificatori (nomi) usano lettere maiuscole

– Ogni procedura è associata ad un numero di procedura unico all'interno di un programma (nell'esempio 1)

Implementazione del programma RPC

- Il programmatore deve sviluppare:
  - il **programma client**: implementazione del main() e della logica necessaria per reperimento e binding del servizio/i remoto/i (esempio: square_client.c)
  - il **programma server**: implementazione di tutte le procedure (servizi) (esempio: square_server.c)
- Attenzione: il programmatore **non** realizza il main() lato server...
  - Chi invoca la procedura remota (lato server)?

RPC: un primo esempio

- Esempio: quadrato di un numero intero
- Esaminiamo il codice della tradizionale procedura **locale**

Esempio: square_local.c

```
#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);
}
```

RPC: un primo esempio (2)

- Procedura **locale**: quadrato di un numero intero *(continua)*

```
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);
}
```

Come si trasforma nel caso di procedura remota?

Sviluppo della procedura di servizio

- Il codice di servizio della procedura remota è quasi identico alla procedura locale

```
#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);
}
```

Esempio: server.c

- Osservazioni:
  - I parametri di ingresso e uscita vengono passati per riferimento
  - Il risultato punta ad una variabile statica (allocazione globale), per essere presente anche oltre la chiamata della procedura
  - Il nome della procedura cambia leggermente (si aggiunge il carattere underscore seguito dal numero di versione, tutto in caratteri minuscoli)

Sviluppo della procedura di servizio (2)

- Queste versioni di server.c non sono corrette: perché?

```
…
square_out *squareproc_1_svc(square_in *inp, struct svc_req *rqstp) {
  square_out *outp;

  outp = (square_out *)malloc(sizeof(square_out));
  outp->res1 = inp->arg1 * inp->arg1;
  return(outp);
}
```

```
…
square_out *squareproc_1_svc(square_in *inp, struct svc_req *rqstp) {
  square_out *outp;

  free(outp);
  outp = (square_out *)malloc(sizeof(square_out));
  outp->res1 = inp->arg1 * inp->arg1;
  return(outp);
}
```


Sviluppo della procedura di servizio (3)

- Questa versione di server.c è corretta: perché?

```
…
square_out *squareproc_1_svc(square_in *inp, struct svc_req *rqstp) {
  static square_out *outp;

  free(outp);
  outp = (square_out *)malloc(sizeof(square_out));
  outp->res1 = inp->arg1 * inp->arg1;
  return(outp);
}
```

Sviluppo del programma client

- Il client viene invocato col nome dell'host remoto ed il valore intero e richiede il servizio di square remoto

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

CLIENT *clnt_create(const char *host, unsigned long program, unsigned number versnum, const char *protocol)

Esempio: client.c

Sviluppo del programma client (2)

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

Sviluppo del programma client (3)

- Notare la creazione del gestore di trasporto client: il gestore di trasporto (cl) gestisce la comunicazione col server, nell'esempio utilizzando un trasporto con connessione ("tcp")
- Il client deve conoscere:
  - l'host remoto dove è in esecuzione il servizio
  - alcune informazioni per invocare il servizio stesso (programma, versione e nome della procedura)
- Per la chiamata della procedura remota:
  - Il nome della procedura cambia: si aggiunge il carattere underscore seguito dal numero di versione (in caratteri minuscoli)
  - I parametri di ingresso della procedura chiamata sono 2:
    - il parametro di ingresso vero e proprio
    - il gestore di trasporto del client
- Il client gestisce gli errori che si possono presentare durante l'invocazione remota usando le funzioni di stampa degli errori: clnt_pcreateerror() e clnt_perror()

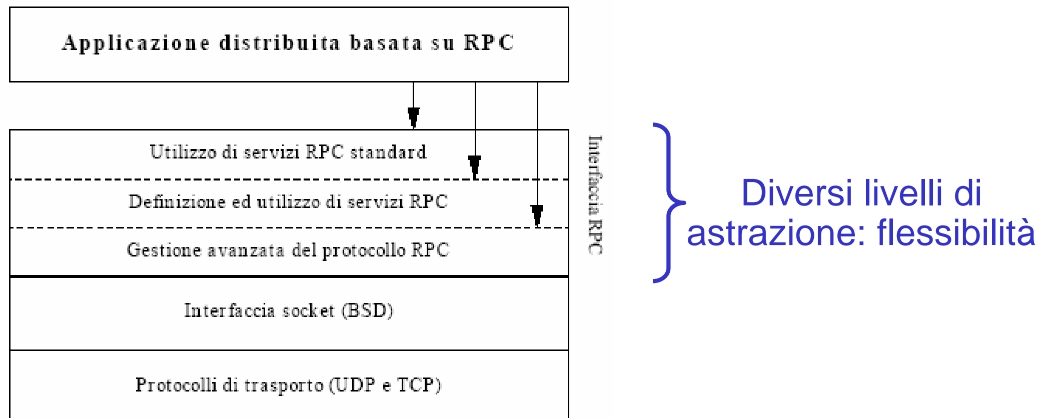
Passi di base per lo sviluppo di SUN RPC

1. Definire servizi e tipi di dati (se necessario) → square.x
2. Generare in modo automatico gli stub del client e del server e (se necessario) le funzioni di conversione XDR → uso di RPCGEN
   - Genera square_**clnt.c**, square_**svc.c**, square_**xdr.c**, square**.h**
3. Realizzare i programmi client e server (client.c e server.c), compilare tutti i file sorgente (programmi client e server, stub e file per la conversione dei dati) e fare il linking dei file oggetto
4. Pubblicare i servizi (lato server)
   1. Attivare il portmapper
   2. Registrare i servizi presso il portmapper (attivando il server)
5. Reperire (lato client) l'endpoint del server tramite il portmapper e creare il gestore di trasporto per l'interazione col server

   A questo punto l'interazione fra client e server può procedere

RPC e socket

- Le RPC sfruttano i servizi dei socket:
  - TCP per stream
    - servizi con connessione
  - UDP per datagram (default in SUN RPC)
    - servizi senza connessione



Caratteristiche di SUN RPC

- Un programma tipicamente contiene più procedure remote
  - Previste anche versioni multiple delle procedure
  - Un unico argomento in ingresso ed in uscita per ogni invocazione (passaggio per copia)
- Mutua esclusione garantita dal programma (e server): di default non si prevede alcuna concorrenza nell'ambito dello stesso programma server
  - Server sequenziale ed una sola invocazione eseguita per volta
  - Per server multithreaded (su SunOS ma non su Linux): rpcgen con opzioni –M e –A
- Fino al ritorno al programma cliente, il processo client è in modo sincrono bloccante in attesa della risposta
- Semantica at-least-once della comunicazione
  - Di default vengono fatte un numero di ritrasmissioni dopo un intervallo di time-out

Identificazione di messaggi RPC

- Per l'identificazione globale un messaggio di richiesta RPC deve contenere
  - numero di programma
  - numero di versione
  - numero di procedura
- Numeri di programma (32 bit) appartenenti a:

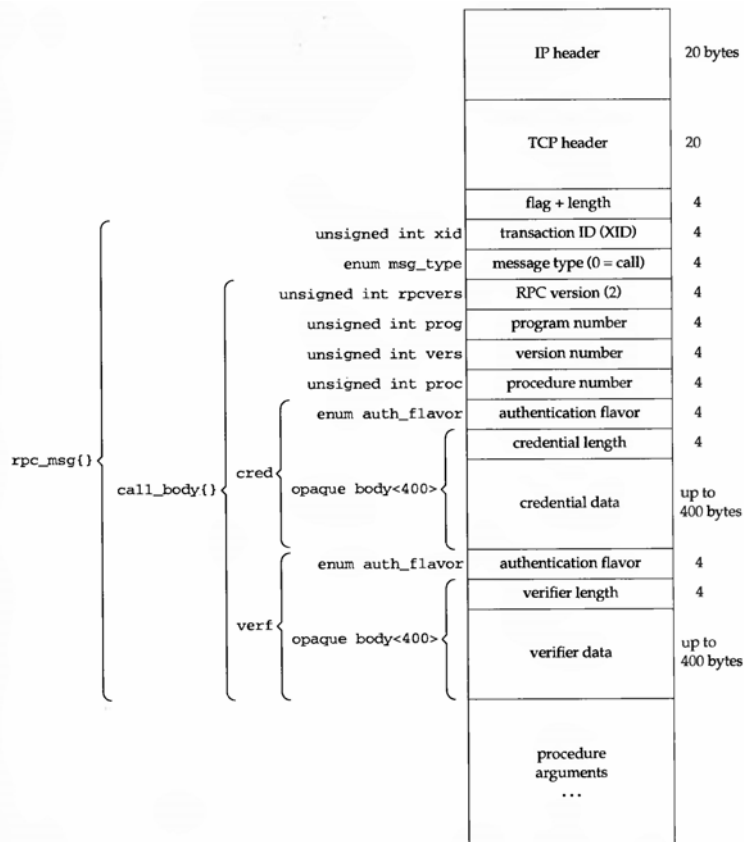
  [0x00000000, 0x1fffffff]: predefinito da Sun; per applicazioni d'interesse comune

  [0x20000000, 0x3fffffff]: definibile dall'utente

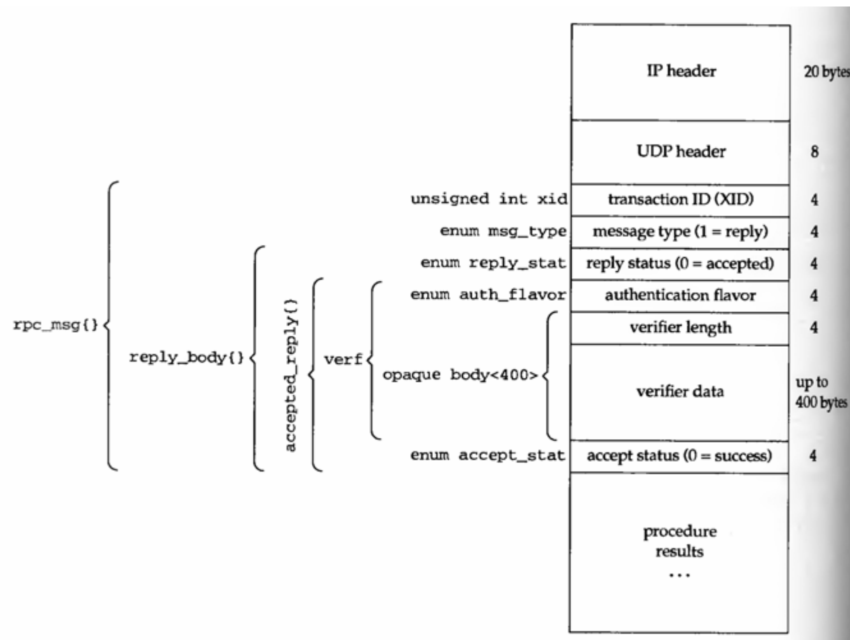
  [0x40000000, 0x5fffffff]: transitorio, riservato alle applicazioni

  [0x6fffffff, 0xffffffff]: riservato per estensioni
- Inoltre, 16 bit per il numero di porta su cui il server risponde

Messaggio di richiesta RPC su TCP



Messaggio di risposta RPC su UDP

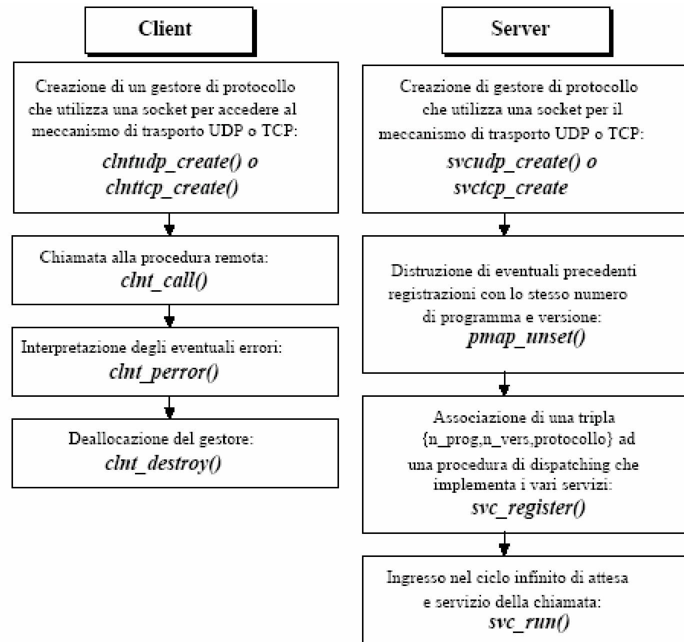


Livelli di SUN RPC

- Livello alto: utilizzo di servizi RPC standard
  - Esempi di procedure pronte per essere invocate: rnusers(), rusers(), rstat()
  - Numeri di programmi noti
- Livello intermedio: definizione ed utilizzo di nuovi servizi RPC
  - Due primitive: callrpc() e registerrpc()
  - CLIENT - callrpc(): chiamata esplicita al meccanismo RPC e provoca l'esecuzione della procedura remota
  - SERVER - registerrpc(): registrazione della procedura remota, associando un identificatore unico alla procedura implementata nell'applicazione

Livelli di SUN RPC (2)

- Livello basso: gestione avanzata del protocollo RPC
  - Chiamata remota tramite funzioni avanzate per ottenere massima capacità espressiva



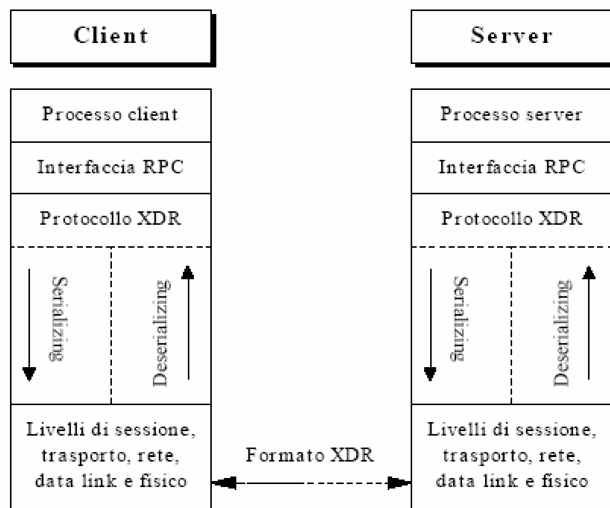
Eterogeneità nella rappresentazione dei dati

- Per comunicare tra host eterogenei 2 soluzioni possibili:

1. Dotare ogni nodo di tutte le funzioni di conversione possibili per ogni rappresentazione dei dati
   - Alte prestazioni e alto numero di funzioni di conversione, pari a N*(N-1), con N numero di host
2. Concordare un formato comune di rappresentazione dei dati: ogni host possiede le funzioni di conversione da/per questo formato
   - Minore prestazioni, ma basso numero di funzioni di conversione, pari a 2*N
   - Soluzione adottata da XDR
     - Ogni host fornisce solamente le proprie funzionalità di trasformazione
       - dal formato locale a quello standard
       - dal formato standard a quello locale

eXternal Data Rapresentation (XDR)

- Funzioni di marshaling svolte dal protocollo XDR
- Funzioni *built-in* di conversione, relative a:
  - Tipi atomici predefiniti
    - xdr_bool(), xdr_char(), xdr_int(), …
  - Tipi standard (costruttori riconosciuti)
    - xdr_array(), xdr_string(), …



Definizione del file.x

- Prima parte del file
  - Definizioni XDR delle costanti
  - Definizioni XDR dei tipi di dati dei parametri di ingresso e uscita per cui per tutti i tipi di dato per i quali non esiste una corrispondente funzione XDR built-in
- Seconda parte del file
  - Definizioni XDR delle procedure
- Esempio in square.x: SQUAREPROC è la procedura numero 1 della versione 1 del programma numero 0x31230000
- Da notare che per le specifiche del protocollo RPC:
  - Il numero di procedura zero (0) è riservato a NULLPROC
  - Ogni definizione di procedura ha un solo parametro d'ingresso e d'uscita
  - Gli identificatori di programma, versione e procedura usano tutte lettere maiuscole ed il seguente spazio di nomi:

    <NOMEPROGRAMMA>PROG per il nome del programma
    <NOMEPROGRAMMA>VERS per la versione del programma
    <NOMEPROCEDURA> per i nomi delle procedure

Server: registrazione procedura remota

- Una procedura registrata può essere invocata: alla procedura viene associato un identificatore strutturato secondo il protocollo RPC
  - Il client deve conoscere il numero di porta su cui il server risponde
- Il server deve registrare il programma RPC nella tabella dinamica dei servizi dell'host su cui il server viene eseguito
- La tabella (detta port map) contiene una registrazione di: tripla {numero_programma, numero_versione, protocollo_di_trasporto} e numero di porta
  - Manca il numero di procedura dato che tutte le procedure RPC all'interno di un programma condividono lo stesso gestore di trasporto
- La gestione della tabella di port map si basa su un processo unico (detto port mapper) per ogni host sui cui è eseguito un server RPC

Port mapper

- Port mapper lanciato come daemon (`portmap`) in ascolto sulla porta 111 (sia UDP che TCP)
- Il port mapper identifica il numero di porta associato ad un programma RPC (servizio) registrato
  - Allocazione dinamica dei servizi sui nodi
- Il portmapper registra i servizi sul nodo e offre le seguenti procedure:
  - Inserimento di un servizio
  - Eliminazione di un servizio
  - Corrispondenza associazione astratta e porta
  - Intera lista di corrispondenza
  - Supporto all'esecuzione remota
- Di default utilizza solo il trasporto UDP
- Limiti: dal client ogni chiamata interroga il portmapper e poi attua la richiesta

Port mapper (2)

- Per avere la lista di tutti i programmi RPC invocabili su di un host usare `rpcinfo -p nomehost`

```
tiberius:~/RPC/square$ rpcinfo -p
 programma vers proto   porta
    100000    2   tcp     111  portmapper
    100000    2   udp     111  portmapper
    100024    1   udp   32768  status
    100024    1   tcp   32769  status
 …
    100007    2   udp     931  ypbind
    100007    1   udp     931  ypbind
    100007    2   tcp     934  ypbind
    100007    1   tcp     934  ypbind
 824377344    1   udp   32825
 824377344    1   tcp   33355
```

824377344 = 0x31230000: numero di programma in square.x

- `/etc/rpc` contiene l'elenco dei programmi RPC disponibili

Processo di sviluppo

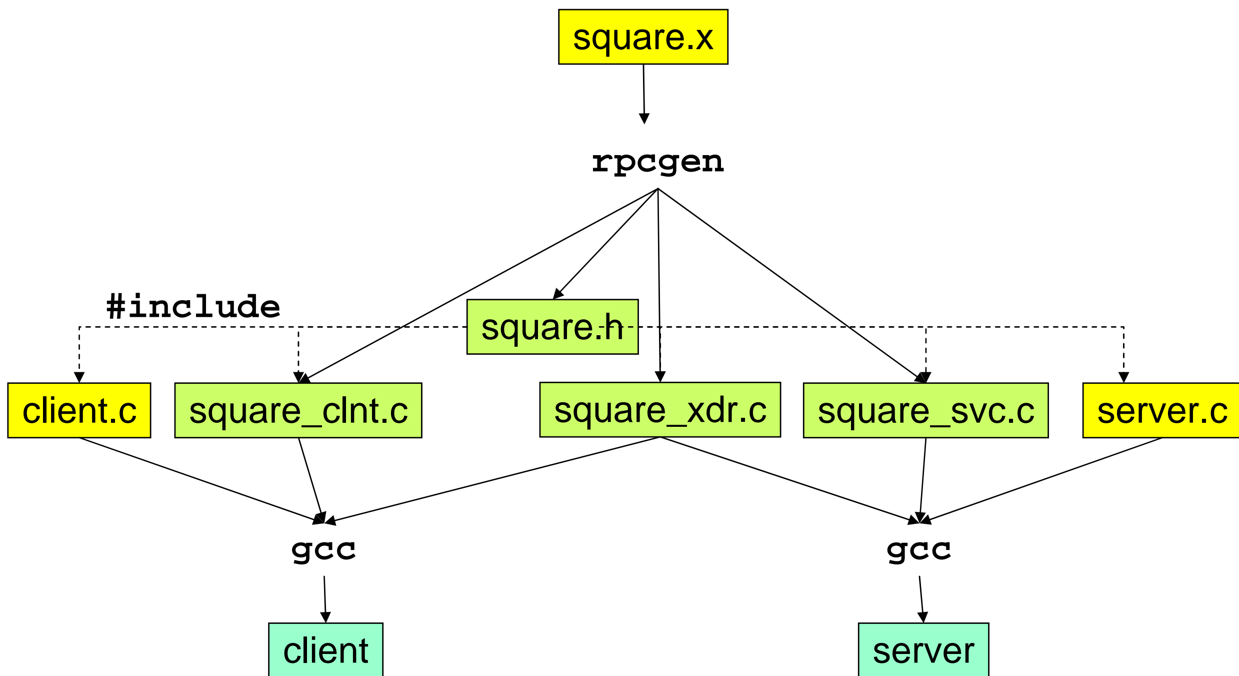
Data una specifica di partenza
- file di linguaggio XDR
  - square.x

RPCGEN produce in automatico
- header file
  - square.h
- file stub del client
  - square_clnt.c
- file stub del server
  - square_svc.c
- file di routine XDR
  - square_xdr.c

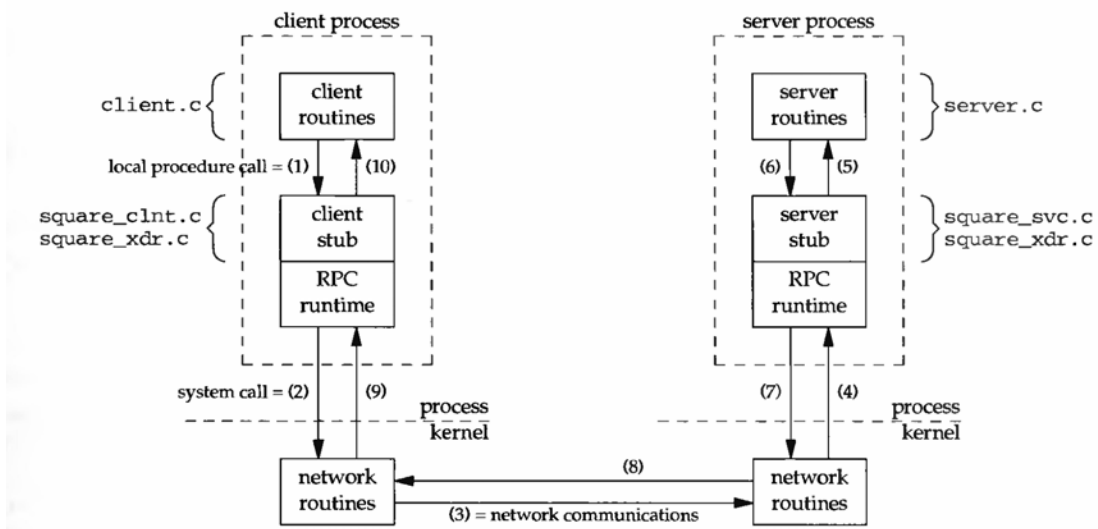
} Analizziamoli

Lo sviluppatore deve realizzare
- programma client
  - client.c
- programma server
  - server.c

Processo di sviluppo (2)



Passi in una RPC



File prodotti da rpcgen: square.h

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#ifndef _SQUARE_H_RPCGEN
#define _SQUARE_H_RPCGEN

#include <rpc/rpc.h>

#ifdef __cplusplus
extern "C" {
#endif

struct square_in {
   long arg1;
};
typedef struct square_in square_in;

struct square_out {
   long res1;
};
typedef struct square_out square_out;

#define SQUARE_PROG 0x31230000
#define SQUARE_VERS 1
```

Il file viene incluso dai due stub generati client e server

In caso di nuovi tipi di dati si devono definire le nuove **strutture dati** per le quali saranno generate in automatico le nuove **funzioni di trasformazione**

File prodotti da rpcgen: square.h (2)

```
#if defined(__STDC__) || defined(__cplusplus)
#define SQUAREPROC 1
extern  square_out * squareproc_1(square_in *, CLIENT *);
extern  square_out * squareproc_1_svc(square_in *, struct svc_req *);
extern int square_prog_1_freeresult (SVCXPRT *, xdrproc_t, caddr_t);

#else /* K&R C */
#define SQUAREPROC 1
extern  square_out * squareproc_1();
extern  square_out * squareproc_1_svc();
extern int square_prog_1_freeresult ();
#endif /* K&R C */

/* the xdr functions */

#if defined(__STDC__) || defined(__cplusplus)
extern  bool_t xdr_square_in (XDR *, square_in*);
extern  bool_t xdr_square_out (XDR *, square_out*);

#else /* K&R C */
extern bool_t xdr_square_in ();
extern bool_t xdr_square_out ();

#endif /* K&R C */

#ifdef __cplusplus
}
#endif

#endif /* !_SQUARE_H_RPCGEN */
```

File prodotti da rpcgen: square_xdr.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */
#include "square.h"

bool_t xdr_square_in (XDR *xdrs, square_in *objp)
{
   register int32_t *buf;

   if (!xdr_long (xdrs, &objp->arg1))
        return FALSE;
   return TRUE;
}

bool_t xdr_square_out (XDR *xdrs, square_out *objp)
{
   register int32_t *buf;

   if (!xdr_long (xdrs, &objp->res1))
        return FALSE;
   return TRUE;
}
```


File prodotti da rpcgen: client stub

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include <memory.h> /* for memset */
#include "square.h"

/* Default timeout can be changed using clnt_control() */
static struct timeval TIMEOUT = { 25, 0 };

square_out *
squareproc_1(square_in *argp, CLIENT *clnt)
{
   static square_out clnt_res;

   memset((char *)&clnt_res, 0, sizeof(clnt_res));
   if (clnt_call (clnt, SQUAREPROC,
        (xdrproc_t) xdr_square_in, (caddr_t) argp,
        (xdrproc_t) xdr_square_out, (caddr_t) &clnt_res,
        TIMEOUT) != RPC_SUCCESS) {
        return (NULL);
   }
   return (&clnt_res);
}
```

Assegnamento timeout per la chiamata

Reale chiamata remota nello stub

Esempio: square_clnt.c

File prodotti da rpcgen: server stub

Esempio: square_svc.c

```
/*
 * Please do not edit this file.
 * It was generated using rpcgen.
 */

#include "square.h"
#include <stdio.h>
#include <stdlib.h>
#include <rpc/pmap_clnt.h>
#include <string.h>
#include <memory.h>
#include <sys/socket.h>
#include <netinet/in.h>
#ifndef SIG_PF
#define SIG_PF void(*)(int)
#endif

static void
square_prog_1(struct svc_req *rqstp, register SVCXPRT *transp)
{
   union {
        square_in squareproc_1_arg;
   } argument;
   char *result;
   xdrproc_t _xdr_argument, _xdr_result;
   char *(*local)(char *, struct svc_req *);
```

Funzione di dispatching

File prodotti da rpcgen: server stub (2)

```
   switch (rqstp->rq_proc) {
   case NULLPROC:
        (void) svc_sendreply (transp, (xdrproc_t) xdr_void,
   (char *)NULL);
        return;
   case SQUAREPROC:
        _xdr_argument = (xdrproc_t) xdr_square_in;
        _xdr_result = (xdrproc_t) xdr_square_out;
        local = (char *(*)(char *, struct svc_req *))
   squareproc_1_svc;
        break;
   default:
        svcerr_noproc (transp);
        return;
   }
   memset ((char *)&argument, 0, sizeof (argument));
   if (!svc_getargs (transp, (xdrproc_t) _xdr_argument,
   (caddr_t) &argument)) {
        svcerr_decode (transp);
        return;
   }
   result = (*local)((char *)&argument, rqstp);
   if (result != NULL && !svc_sendreply(transp, (xdrproc_t)
   _xdr_result, result)) {
        svcerr_systemerr (transp);
   }
```

File prodotti da rpcgen: server stub (3)

```
    if (!svc_freeargs (transp, (xdrproc_t) _xdr_argument,
    (caddr_t) &argument)) {
         fprintf (stderr, "%s", "unable to free arguments");
         exit (1);
    }
    return;
}

int
main (int argc, char **argv)
{
    register SVCXPRT *transp;

    pmap_unset(SQUARE_PROG, SQUARE_VERS);

    transp = svcudp_create(RPC_ANYSOCK);
    if (transp == NULL) {
         fprintf (stderr, "%s", "cannot create udp service.");
         exit(1);
    }
    if (!svc_register(transp, SQUARE_PROG, SQUARE_VERS,
    square_prog_1, IPPROTO_UDP)) {
         fprintf (stderr, "%s", "unable to register
    (SQUARE_PROG, SQUARE_VERS, udp).");
         exit(1);
    }
```

Distruzione di eventuali precedenti registrazioni con lo stesso numero di programma e versione

Creazione di gestore di protocollo che utilizza socket UDP

Associazione della tripla {n_prog, n_vers, protocollo} ad una procedura di dispatching che implementa i vari servizi

File prodotti da rpcgen: server stub (4)

```
    transp = svctcp_create(RPC_ANYSOCK, 0, 0);
    if (transp == NULL) {
         fprintf (stderr, "%s", "cannot create tcp service.");
         exit(1);
    }
    if (!svc_register(transp, SQUARE_PROG, SQUARE_VERS,
    square_prog_1, IPPROTO_TCP)) {
         fprintf (stderr, "%s", "unable to register
    (SQUARE_PROG, SQUARE_VERS, tcp).");
         exit(1);
    }

    svc_run();
    fprintf (stderr, "%s", "svc_run returned");
    exit (1);
    /* NOTREACHED */
}
```

Creazione di gestore di protocollo che utilizza socket TCP

Ingresso nel ciclo infinito di attesa e servizio della chiamata

Esercizio

- Sviluppare un'applicazione client/server basata su SUN RPC per ottenere l'echo di una stringa
- Il client interagisce con l'utente richiedendo una stringa, invoca la procedura remota echo passando come parametro la stringa letta e stampa a video la stringa ottenuta come valore di ritorno della procedura invocata
  - Queste operazioni sono ripetute finché l'utente non interrompe l'interazione
- Il server realizza la procedura di echo, che restituisce come risultato la stringa passata come parametro di ingresso

Riferimenti per RPC

- man rpc
- man rpcgen
- Capitolo 16 di W.R. Stevens, "Unix Network Programming, Volume 2: Interprocess Communication", 1999.
- E. Petron, "Remote Procedure Calls", Linux Journal, 1997