DPIoT - Riassunto

Tommaso Puccetti
Studente presso Universita degli studi di Firenze

November 29, 2019

Contents

List of Tables

List of Figures

1 Communication Mechanisms

1.1 Middleware

Il **middleware** un insieme di applicazioni e protocolli "**general purpose**" che risiedono all'interno del livello applicativo. dunque un livello software che astrae dall'eterogeneit di rete, hardware, sistemi operativi e linguaggi di programmazione, con lo **scopo di fornire interfacce comuni che assicurino modelli di comunicazione e di computazione uniformi**. Questo livello, dunque, costituisce un insieme di protocolli condivisi dalle applicazioni pi specifiche al livello soprastante. In sintesi, un livello middleware offre servizi alle applicazioni quali:

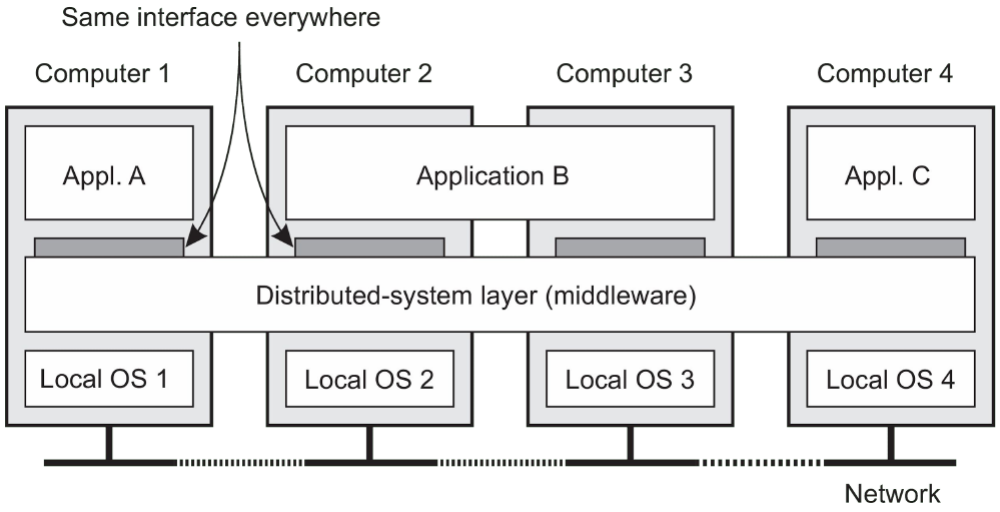

Figure 1: Livello Middleware

- Comunicazione;
- Meccanismi di sicurezza;
- Transazioni
- Error-recovery;
- Gestione di risorse condivise.

Questi servizi sono indipendenti rispetto alle specifiche applicazioni. Alcuni esempi:

- Protocolli di autenticazione e autorizzazione (criptografia ssh)
- Protocolli di commit. Sono utilizzati per realizzare l'atomicit nelle transazioni. Stabiliscono se in un insieme di processi tutti hanno svolto una particolare operazione o se non stata svolta affatto.

Nello specifico vedremo come i **protocolli di comunicazione middleware supportino servizi di comunicazione ad alto livello** e permettano, per esempio, la chiamata a procedure o oggetti remoti in modo **trasparente.**

1.2 Coordinazione diretta

Un tipi di comunicazione nella quale le componenti partecipanti sono:

- **Referentially coupled**: durante la comunicazione gli attori utilizzano riferimenti espliciti ai loro interlocutori.
- **Temporally coupled**: entrambe le componenti devono essere in esecuzione (up and running).

Il libro propone un'introduzione ai tipi di comunicazione (persist, transient, synchronous, asynchronous).

1.3 Remote Procedure Call

Molti sistemi distribuiti sono basati sullo scambio di messaggi tra processi, tuttavia questo tipo di approccio non permette di nascondere la comunicazione tra le componenti in modo da rendere trasparente il contesto distribuito.
Una soluzione al problema stata proposta da Nelson e Birrell (1984) introducendo una modalit completamente differente nella gestione della comunicazione nel contesto di un sistema distribuito. In breve la proposta quella di chiamare procedure che sono localizzate su macchine remote:

1. quando A chiama B il processo chiamante in A sospeso;
2. l'esecuzione della procedura chiamata ha luogo in B;
3. A invia i parametri della chiamata a B che a sua volta risponder con il risultato della chiamata;
4. **Nessun passaggio di messaggi visibile dal punto di vista del programmatore.**

La soluzione ha le seguenti problematiche:

- le procedure chiamante e chiamato si trovano su macchine diverse e non condividono lo stesso address space;
- la rappresentazione dei parametri e del risultato di ritorno pu differire sulle macchine interessate;

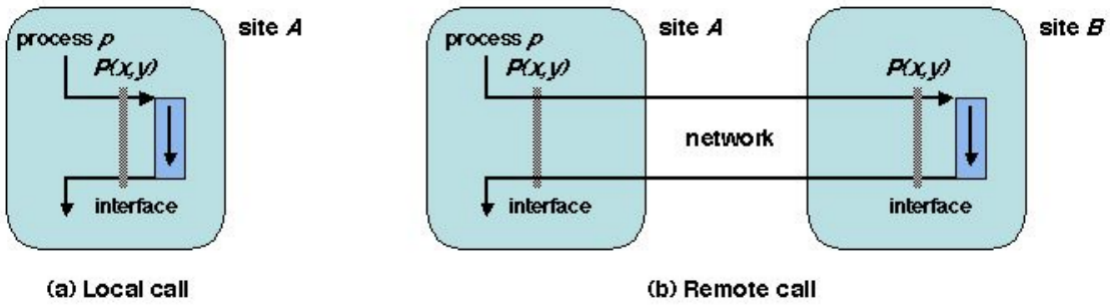

Figure 2: Chiamata a procedura locale vs remota

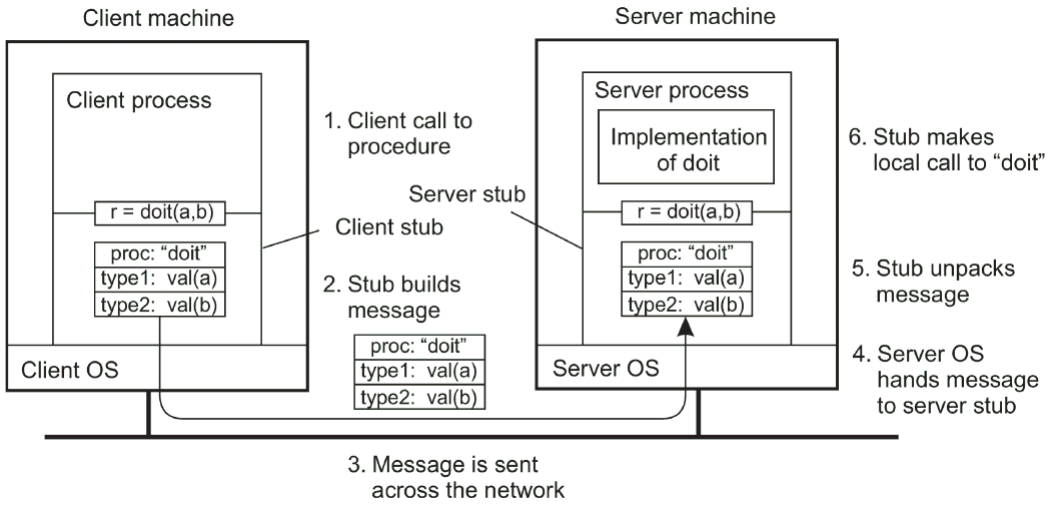

Figure 3: Funzionamento RPC

- Le due macchine potrebbero crashare.

Una chiamata a procedura remota deve essere **trasparente** rispetto al chiamante, per farlo viene creato uno stub locale della funzione che si trova in macchina remota. Lo stub, sia sul server che sul client implementa serializzazione e invio dei parametri e del risultato. Di seguito si elencano i passi necessari ad una chiamata a procedura remota:

1. la procedura del client chiama il proprio stub;
2. lo stub costruisce il messaggio ed effettua una chiamata al proprio OS;

3. l'OS del client invia il messaggio all'OS remoto;
4. l'OS remoto invia il messaggio allo stub del server;
5. lo stub del server decomprime i parametri e chiama la procedura locale sul server;
6. si esegue la computazione e si invia i risultati allo stub;
7. lo stub del server comprime i risultati e li invia al proprio OS;
8. si invia il messaggio all'OS del client che lo passa allo stub del client;
9. lo stub decomprime il risultato della computazione e lo passa al client

1.3.1 Passaggio di parametri

L'operazione di impacchettare parametri all'interno di un messaggio chiamata **marhaling**, il messaggio conterr i parametri stessi e le informazioni necessarie al destinatario. Il principale problema il seguente: **client e server potrebbero adottare diverse rappresentazioni per i dati** (esempio diverse little endian big endian). Nel caso di utilizzo di HTTP come protocollo di trasporto il formato xml pu essere utilizzato come formato comune per il passaggio dei parametri.

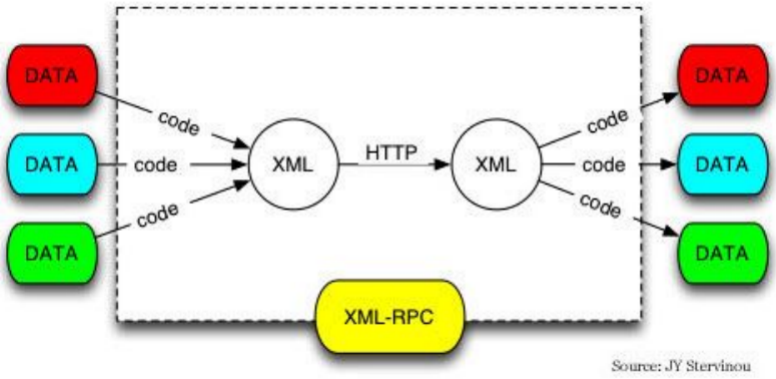

Figure 4: Xml

Un problema ulteriore risiede nel **passaggio dei puntatori e riferimenti**. Infatti, questi avranno senso solo se riferiti allo spazio di indirizzi locale del chiamante. Una possibile soluzione quella di sostituire la **chiamata per riferimento** con un **copia/ripristina**. L'idea quella di effettuare una copia dell'array da passare ed allegarla al messaggio destinato al server. L'array conservato in un buffer nello stub del server ed inviato nuovamente al client una volta effettuata la chiamata remota (se richiesto). Nonostante i linguaggi offrano supporto automatico al **(un)marshaling**, quest'ultimo introduce un'**overhead** nella comunicazione, soprattutto in caso di grosse strutture dati come alberi e grafi.

```
# Stub on the client
class Client:
  def append(self, data, dbList):
    msglst = (APPEND, data, dbList)
    msgsnd = pickle.dumps(msglst)
    self.chan.sendTo(self.server, msglst)
    msgrcv =
self.chan.recvFrom(self.server)
    return msgrcv[1]
```

```
# Main loop of the server
while True:
  msgreq = self.chan.recvFromAny()
  client = msgreq[0]
  msgrpc = pickle.loads(msgreq[1])
  if APPEND == msgrpc[0]:
    result = self.append(msgrpc[1],
msgrpc[2])
    msgres = pickle.dumps(result)
    self.chan.sendTo(client, result)
```

Figure 5: Marshaling in Java

Il problema non si presenta qualora i riferimenti siano **globali**, ovvero quando hanno un significato sia per il server sia per il client.
In generale, nel contesto di un sistema basato sugli oggetti sono definite due tipologie di oggetti:

- **Locali**: copiati e trasmessi nella loro interezza;
- **Remoti**: solo lo stub copiato e trasmesso.

In Java oggetti remoti o locali hanno tipi diversi (i remoti implementano l'interfaccia Remote).

1.3.2 Implementare RPC

Ci sono due modi attraverso il quale il meccanismo RPC pu essere fornito allo sviluppatore:

- **Framework o libreria**: il programmatore deve specificare cosa esportato in remoto fornendo di fatto un'**interfaccia del servizio**, che

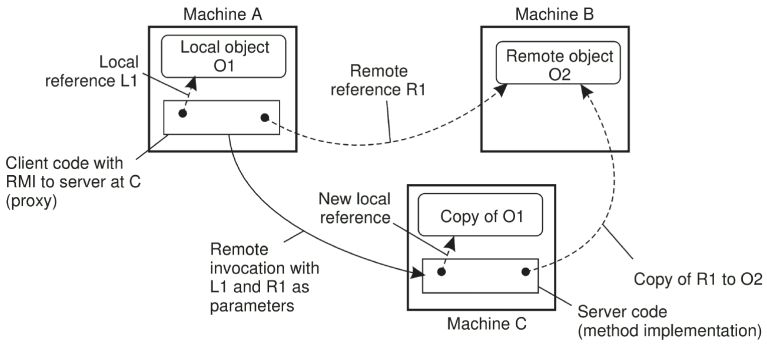

Figure 6: Oggetti remoti e locali

contiene tutte le procedure che possono essere chiamate dal client. I framework hanno il pregio di essere **indipendenti dal linguaggio**. Per questo norma utilizzare un **Interface Definition Language (IDL)** che, una volta compilato, genere gli stub per client e server nel linguaggio desiderato. Di contro non abbiamo trasparenza totale per il programmatore che dunque consapevole di trovarsi nel contesto di una chiamata a procedura remota (deve specificare egli stesso gli oggetti remoti). Alcuni esempi di framework: **Corba, GRPC, Apache Thrift**.

- **Costrutti all'interno del linguaggio**: lo stesso linguaggio a definire i costrutti necessari ad una RPC. In questo caso il **compilatore a generare gli stub** per client e server. In questo modo si ottiene **trasparenza** per il programmatore, tuttavia client e server devono essere **implementati nello stesso linguaggio** (Es: **Java RMI**).

1.3.3 RPC Asincrono

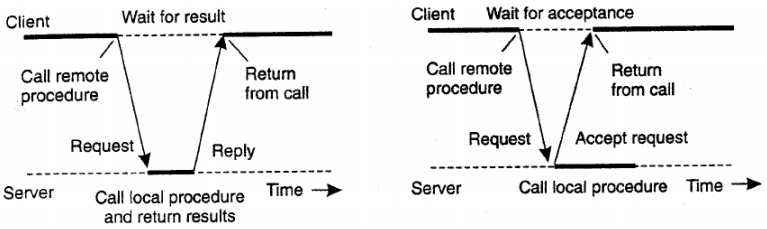

Figure 7: RPC tradizionale e asincrona

A differenza del paradigma tradizionale nel quale il client attende la risposta del server bloccando la sua esecuzione, il server invia un ACK al client una volta ricevuta la richiesta. L'ACK viene inviato al client per notificare che la sua richiesta sar processata, nel frattempo il client pu eseguire ulteriori operazioni evitando di sospendere la sua esecuzione. Il Server utilizza una funzione detta di **Callback** per consegnare il risultato al Client. L'asincronicit della comunicazione permette l'implementazione di un proto-

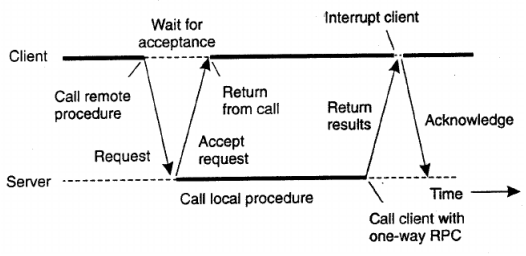

Figure 8: Callback

collo **Multicast RPC** inviando richieste in parallelo a server diversi che dunque processano indipendentemente l'uno dall'altro. Si pu definire questo protocollo nell'ottica di accettare il risultato pi veloce scartando dunque gli altri, oppure per la realizzazione di una computazione distribuita, combinando i risultati ricevuti.

1.3.4 Binding

In applicazioni reali abbiamo bisogno di una fase preliminare chiamata **binding** che permette al client di avere un riferimento al server. Necessario per il client risulta l'utilizzo di un **registro** al cui interno sono salvate coppie (nome, indirizzo) di uno o pi server. Si utilizza tale riferimento per la comunicazione.

1.4 Message Oriented Middleware

Questo modello di comunicazione prevede lo scambio di messaggi tra le entit partecipant. Grazie allo scambio di messaggi possiamo definire un modello nel quale, mittente e destinatario **non devono essere attivi durante lo scambio dei messaggi**. Questo possibile grazie al Middleware che mette a disposizione buffer temporanei per i messaggi scambiati. Ogni applicazione

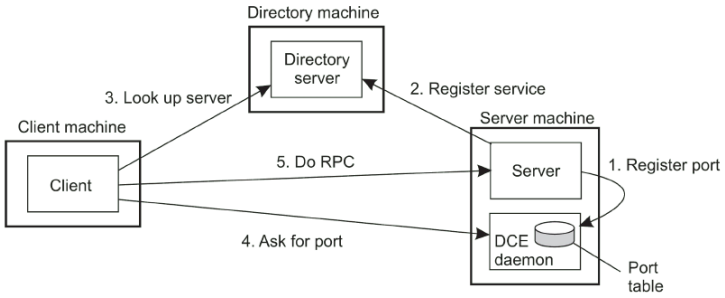

Figure 9: Binding

ha a disposizione una coda locale che contiene i messaggi inviati e ricevuti e che pu eventualmente essere condivisa tra pi applicativi. Il modello di

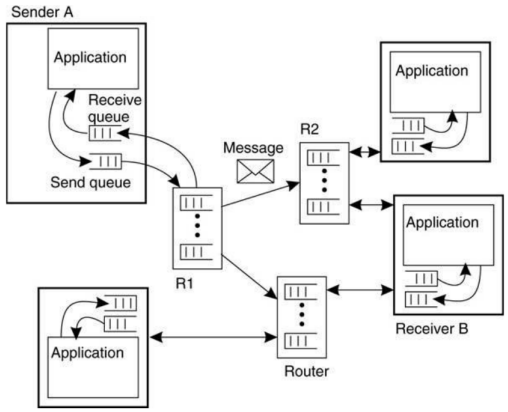

Figure 10: Code

comunicazione definito ha le seguenti propriet:

- La comunicazione avviene semplicemente inserendo e rimuovendo messaggi dalla coda, un messaggio ovviamente rimane nella coda fino a che non esplicitamente rimosso;
- La comunicazione **loosely coupled**, cio significa che il ricevente non deve essere necessariamente in esecuzione.

Di seguito sono elencate le primitive concettuali che un message oriented middleware deve esporre:

- **Put**: inserisce un messaggio nella coda;
- **Get**: rimuove il primo messaggio dalla coda (blocking);
- **Poll**: rimuove il primo messaggio dalla coda (non-blocking);
- **Notify**: informa che un messaggio arrivato nella coda.

1.4.1 Queue Manager

Il queue manager gestisce i messaggi inviati o ricevuti da un'applicazione nella sua coda (ad ogni applicazione associata una coda e un relativo manager). Pu essere implementato come una libreria collegata all'applicazione o come un **processo separato**. *Nel secondo caso il sistema supporter la comunicazione asincrona persistente.* In definitiva questi processi operano come

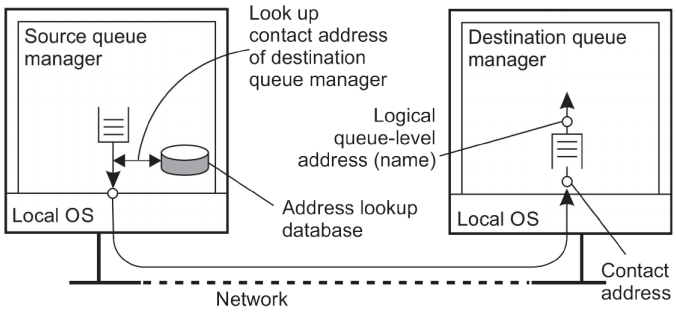

Figure 11: Queue Manager

router o **relay** inoltrando i messaggi ricevuti ad altri queue manager. In questo modo il sistema di quequing pu costituire **un livello applicazione a se stante (Overlay network)** (un'astrazione), basato su una rete di computer esistente. Questa overlay network deve essere collegata e per farlo

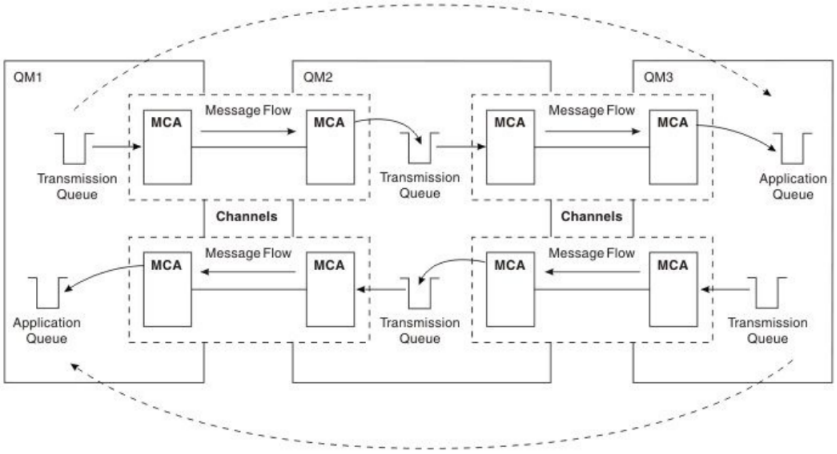

Figure 12: Overlay network

ogni entit deve essere a conoscenza degli indirizzi fisici associati ai nomi delle macchine partecipanti la rete e quindi delle loro rispettive code. Questo approccio **non risulta scalabile** e nel contesto di reti di grosse dimensioni porta ad evidenti **problemi gestionali**. Possiamo migliorare il modello di comunicazione delegando ai router la responsabilit di tenere traccia della topologia di rete e di aggiornare i binding (nome, indirizzo), mentre le altre entit partecipanti possiedono dei riferimenti statici al/ai router pi vicino.

1.4.2 Eterogeneit: Message Brokers

I sistemi distribuiti possono essere eterogenei rispetto ai linguaggi utilizzati per realizzare le singole entit partecipanti. In questi casi difficile definire un protocollo condiviso poich assente alla base un'accordo sul formato dei dati messaggi scambiati.
Un **Message Broker** si comporta come un getway: si occupa di convertire i messaggi ricevuti in un formato consono a quello del ricevente. Nella pratica un message broker usa un repository di regole e programmi che permettono la conversione di un messaggio T1 in uno T2. Esempi di message brokers:

1.5 Java RMI

Java RMI (**Remote Method Invocation**) un framework che permette di implementare il modello RPC nel constesto di un sistema distribuito. Il

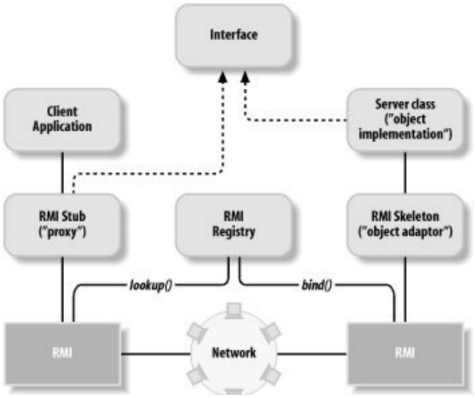

Figure 13: Architettura di RMI

modello presenta 4 entit principali:

- **Interfaccia**: utilizzata per definire la risorsa remota;
- **Server**: implementa la risorsa remota (che sar richiesta dal client);
- **Client**: richiede al server la risorsa remota.
- **Registro**: si occupa di gestire l'accesso alla risorsa remota.

Il **Registro**: un servizio di **naming** che mappa i nomi simbolici degli oggetti remoti al loro stub. Il Server pu registrare un oggetto remoto nel registro scrivendone il nome e l'indirizzo al quale reperibile. Il client cerca l'oggetto remoto all'interno del registro.
L'**interfaccia** specifica un contratto, ovvero le firme dei metodi che si possono invocare sull'oggetto remoto e che dunque ne regolano le modalit di utilizzo. **Per ogni oggetto** che vogliamo rendere accessibile attraverso la rete dobbiamo definire un'interfaccia che estenda l'interfaccia remota ***java.rmi.remote***.
Le interfacce cos definite dal server devono essere note anche al client in modo tale che egli possa operare sugli oggetti ricevuti dal server senza incorrere in errori di tipo. L'interfaccia remota serve solo ad indicare la possibilit di reperire gli oggetti che estendono tale interfaccia in remoto.
Vediamo quali sono i passi per implementare un **RMI server**:

1. Implementare la classe remota definendo costruttore e metodi remoti (estendiamo la classe ***java.rmi.server.UnicastRemoteObject*** e ne chiamiamo il costruttore per esportare l'oggetto);
2. Creare un'istanza dell'oggetto remoto;
3. Registrare tale oggetto remoto all'interno del registro. Per fare questo dobbiamo scegliere un identificativo unico (una stringa) per l'oggetto, che deve essere noto anche al client. Una volta ottenuto un riferimento al registry creiamo un binding tra quel nome e l'istanza dell'oggetto relativa. La classe ***LocateRegistry*** permette di ottenere il riferimento al registro remoto o di crearne uno in ascolto sulla porta desiderata sullo stesso host del server (***createRegistry(int port), getRegistry(String host, int port)***).

Per quanto riguarda il client i passi per l'implementazione sono i seguenti:

1. Localizzare il registro (stessi metodi della classe **LocateRegistry** indicati per il server);

2. Utilizzare un nome simbolico per cercare l'oggetto remoto all'interno del registro;

3. utilizzare l'oggetto remoto chiamandone i metodi.

Possiamo utilizzare RMI per implementare una comunicazione **sincrona** (il client aspetta fino al termine dell'invocazione remota). Possiamo ottenere una comunicazione **asincrona** utilizzando le **callback** il client invoca un oggetto remoto e passa la callback al server (un altro oggetto remoto).

1.6 gRPC

gRPC un framework open source per l'implementazione del modello RPC:

- Si basa su i meccanismi di streaming messi a disposizione da **HTTP/2**;
- **Supporta molti linguaggi** grazie all'utilizzo di un **IDL** (Interface Definition Language).
- Si appoggia a **Protocol Buffer** che un meccanismo per la serializzazione di strutture dati basato su un particolare formato binario che rendono i payload leggeri e veloci da trasmettere. Mette a disposizione un linguaggio proprio utilizzabile per definire interfacce indipendenti dal linguaggio.

I servizi messi a disposizione sono 4:

- **Unary RPCs**: Implementa uno scambio di messaggi **sincrono**.
- **Server streaming RPCs**: Un client invia richieste al server e riceve uno stream di messaggi (il client legge dallo stream fino a che non ci sono piu messaggi).
- **Client streaming RPCs**: Il client scrive una sequenza di messaggi e li manda al server utilizzando uno stream.
- **Bidirection streaming RPCs**: Entrambi i lati della comunicazione utilizzano uno stream in lettura/scrittura per inviare e ricevere messaggi.

Il Workflow di gRPC il seguente:

- **Definire un'interfaccia** utilizzando il Protocol Buffer Language ed il suo IDL (file di testo in formato **.proto**);
- **Compilare** l'interfaccia per ottenere gli stub per client e server e le classi necessarrie alla serializzazione (si utilizza il comando **protoc**).
- **Integrare** gli stub con codice ad-hoc.

```
// The greeter service definition.
service Greeter {
 // Sends a greeting
 rpc SayHello (HelloRequest) returns (HelloReply) {}
}
```

Figure 14: Definizione di un interfaccia con gRPC IDL

FINIRE SU SLIDE

2 Basic distributed algorithms

2.1 Contesto

Il **contesto** nel quale operiamo chiamato **ambiente distribuito**. Consiste in una collezione finita ϵ di **entit** che comunicano attraverso **messaggi** con lo scopo di raggiungere un **obiettivo comune**. Vediamo quali sono le componenti principali del modello:

- **Entit**: l'unit computazionale di un ambiente distribuito, pu essere vista come un processo, un agente, uno switch ecc. Ogni entit equipaggiata con una memoria privata e non condivisa. La memoria composta da un insieme di registri, tra i quali spiccano lo **status register**, che pu assumere i valori di *idle, Processing, Waiting*, e l'**input value register**. Inoltre possibile settare un **alarm clock** locale che pu essere resettato all'occorrenza.
- **Eventi esterni**: Il comportamento di un'entit reattivo ed innescato da stimoli esterni. Questi possono essere:
  - L'arrivo di un messaggio;

- Lo scadere dell'alarm clock;
- Impulsi spontanei.

L'ultimo l'unico stimolo originato da forze che sono esterne al sistema (come esempio si riporta la richiesta ad un bancomat da parte dell'utente nel sistema ATM server- ATM client)

- **Azioni**: un'entit pu svolgere le seguenti **operazioni**:
  - Operazioni sulla memoria locale;
  - Trasmissione dei messaggi;
  - (re)set dell'alarm clock;
  - Cambiare il valore del registro di stato.

  Le azioni sono **atomiche** (non possono essere interrotte) e **finite** (devono terminare in tempo finito). L'azione speciale **nil** permette ad un'entit di non reagire ad uno specifico evento.

- **Comportamenti delle entit**: l'insieme $B(x)$ una funzione $StatoxEvento \rightarrow Azioni$ ovvero una funzione che ad una coppia stato-evento associa un comportamento (pu definire un insieme di comportamenti **deterministico** o **non deterministico**). Un sistema detto **simmetrico** se tutte le entit hanno lo stesso comportamento ($B(x) = B(y) \forall x, y \in E$). Tutti i sistemi possono essere resi simmetrici. **Comunicazioni:** guarda figure.

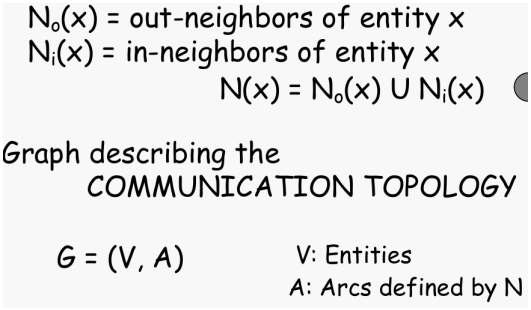

Figure 15: Come rappresentare la topologia di rete

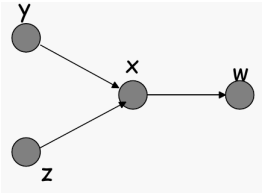

Figure 16: Topologia

2.1.1 Assiomi

- **Delay trasmissione messaggi**: in assenza di **fallimenti** un messaggio inviato da x ad un suon vicino y arriva in un tempo finito.
- **Orientamento Locale**: ogni entit pu distinguere i suoi **out-neighbors** (si utilizzano delle etichette sugli archi). Nella pratica un'entit sa da quale porta il messaggio gli stato recapitato.

Figure 17: Labels

2.1.2 Restrizioni

Si possono definire ulteriori propriet o capacit in relazione ai compiti e agli obiettivi che il sistema distribuito si prepone di raggiungere. Tuttavia queste propriet aggiuntive limitano l'applicabilit reale del protocollo e dunque nella pratica rappresentano delle **restrizioni**. Vediamone alcune:

- **Ordine dei messaggi:** in assena di fallimenti, messaggi trasmessi nello stesso link arrivano nell'ordine d'invio.
- **Link bidirezionali**: $\forall x\ N_i(x) = N_o(x)$ e $\forall y\ \lambda_x(x,y) = \lambda_x(y,x)$
- **Fault detection**:
  - **Edge Failure Detection**: un'entit pu individuare il fallimento di uno dei suoi link;

  - **Entity Failure Detection**: un'entit pu rilevare il fallimento di uno dei suoi vicini

- **Reliability restrinction**:
  - **Guaranteed delivery**: ogni messaggio inviato viene recapitato al mittente non corrotto;
  - **Partial reliability**: garantisce l'assenza di fallimenti in futuro;
  - **Total reliabilit**: non ci sono stati fallimenti e non ce ne saranno.
- **Strongly connected**: il grafo g che rappresenta la topologia fortemente connesso.
- **Knowledge restrinction**
  - conoscenza del numero di nodi;
  - conoscenza del numero di link;
  - conoscenza del diametro.

2.1.3 Tempo ed Eventi

Un evento esterno genera un'azione che dipende dallo stato dell'entit in questione. Un'azione pu a sua volta generare un evento (per esempio l'operazione send genera un evento receiving). Un'ulteriore considerazione riguarda la possibilit che eventi generati in questo modo possano non occorrere nel caso in cui vi sia un fallimento del link di comunicazione. Ovviamente questi eventi se occorrono occorrono dopo del tempo (alla ricezione del messaggio per esempio). Eventi come **receiving** hanno un **delay non predicibile**. Un'esecuzione descritta completamente dalla sequenza di eventi che occorsa. Delay diversi porteranno ad esecuzioni differenti e dunque a risultati possibilmente diversi. Per convenzione tutti gli eventi spontanei sono generati al tempo $t = 0$ prima che l'esecuzione abbia inizio.
Definito $\alpha(x, t)$ lo stato del nodo x al tempo t, importante evidenziare che:

- se un evento avviene in due esecuzioni diverse e gli stati $\alpha 1$ e $\alpha 2$ sono uguali, allora **il nuovo stato interno sar lo stesso in entrambe le esecuzioni**.
- se un evento avviene al tempo t nei nodi x e y ed i loro stati $\alpha(x)$ e $\alpha(y)$ sono uguali, allora i nuovi stati di x e y saranno lo stesso stato.

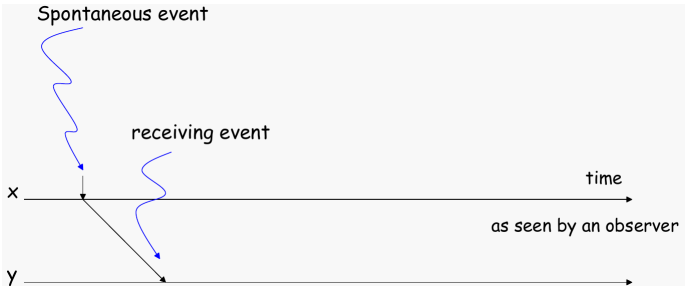

Figure 18: Stato x Evento

2.1.4 Livelli di Conoscenza

- **Local knowledge**: $p \in LK_t[x]$ dove p il contenuto della memoria locale di un'entit e tutte le informazioni derivabili da essa.
- **Implicit knowledge**: $p \in IK_t[W] \_ iff \exists x \in W \ (p \in LK_t[x])$
- **Explicit knowledge**: $p \in EK_t[W] \_ iff \forall x \in W \ (p \in LK_t[x])$

2.2 Broadcast

Considerato un sistema distribuito nel quale solo il nodo x sia a conoscenza di una qualche informazione importante, il **problema del broadcast** consiste nel propagare questa informazione a tutti gli altri nodi. Una soluzione del problema deve essere valida a prescindere dal nodo **iniziatore**.

2.2.1 Flooding

Una soluzione al problema del broadcast data dall'algoritmo **flooding**. L'idea molto semplice: se un nodo a conoscenza di qualcosa invia l'informazione ai suoi vicini. L'algoritmo riportato in figura nella variante per la quale il mittente viene escluso dalla lista dei nodi ai quali inoltrare l'informazione ricevuta. L'algoritmo gode della proprit di **Termination**: l'algoritmo termina in tempo finito (local termination quando lo stato *done*). Garantita dal fatto che il grafo connesso e che vale la propriet di total reliability. Il caso peggiore si presenta quando il **grafo completo**.

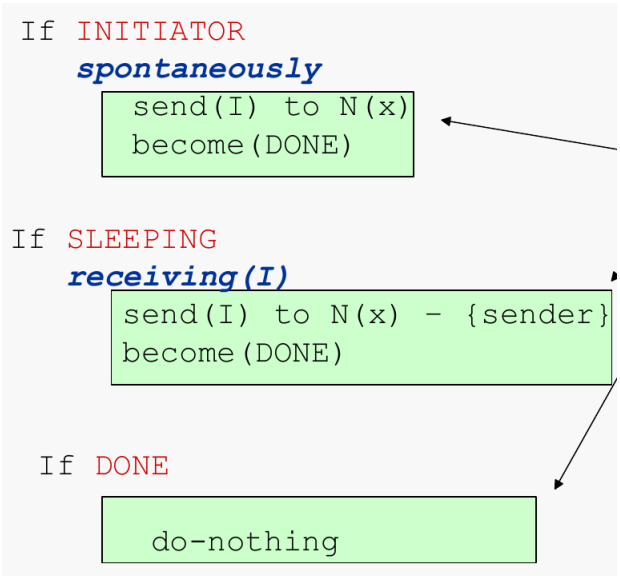

Figure 19: Algoritmo Flooding