DPIoT - Riassunto

Tommaso Puccetti
Studente presso Universita degli studi di Firenze

December 12, 2019

Contents

List of Tables

List of Figures

1 Communication Mechanisms

1.1 Middleware

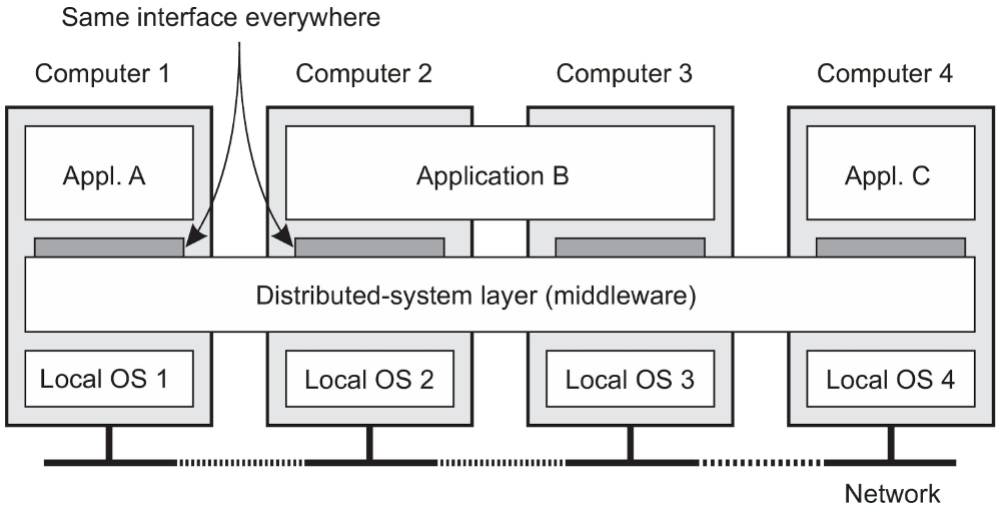

Figure 1: Livello Middleware

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:

- 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;
- Le due macchine potrebbero crashare.

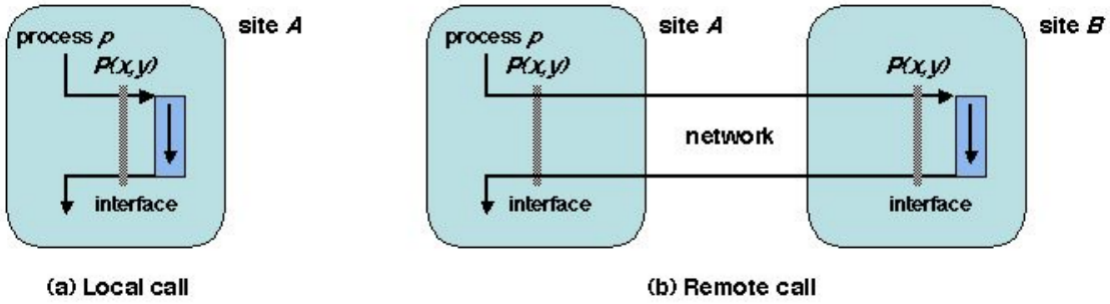

Figure 2: Chiamata a procedura locale vs remota

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;

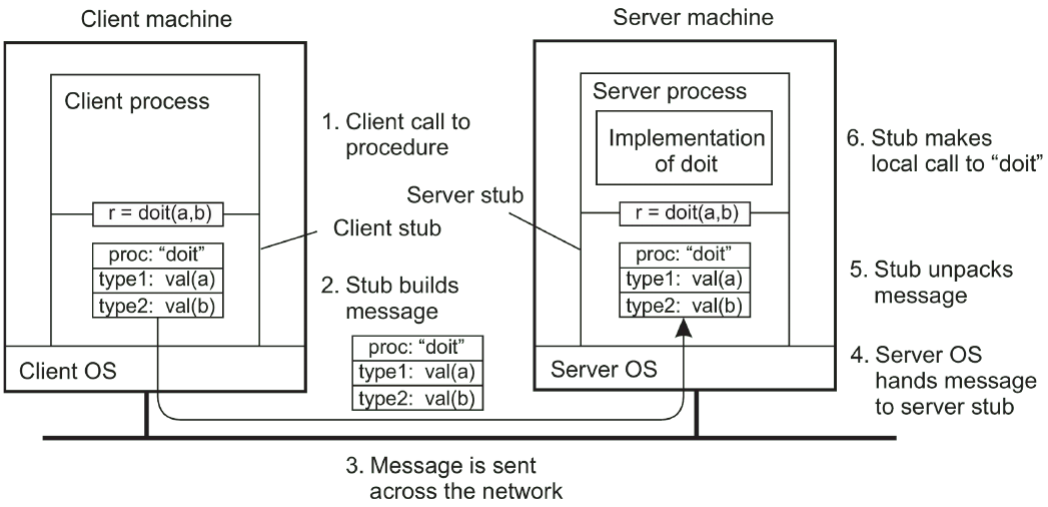

Figure 3: Funzionamento RPC

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.

Un problema ulteriore risiede nel **passaggio dei puntatori e riferimenti**. Infatti, questi avranno senso solo se riferiti allo spazio di indirizzi

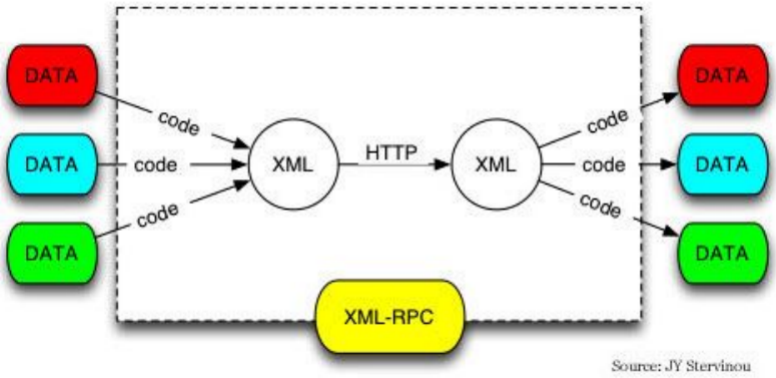

Figure 4: Xml

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).

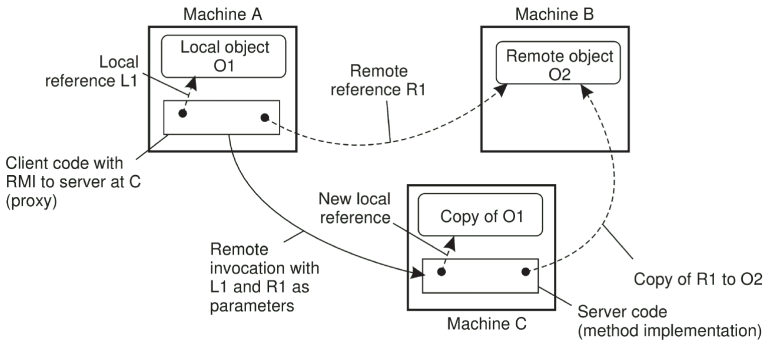

Figure 6: Oggetti remoti e locali

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 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

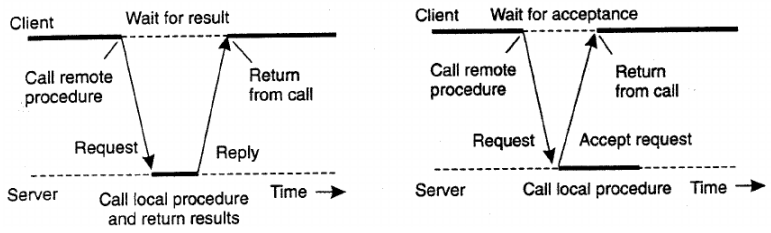

Figure 7: RPC tradizionale e asincrona

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-

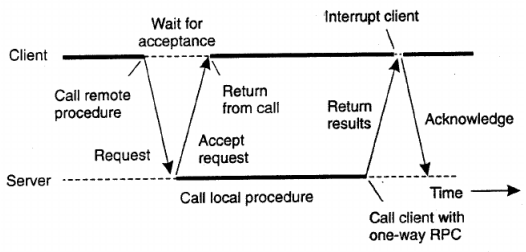

Figure 8: Callback

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.

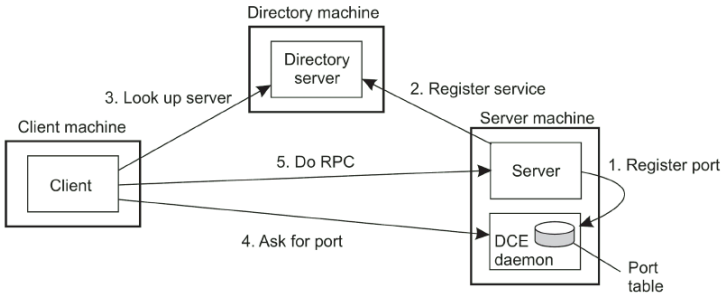

Figure 9: Binding

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 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 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);

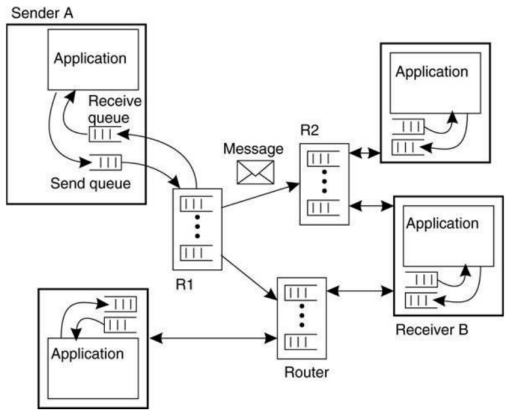

Figure 10: Code

- **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

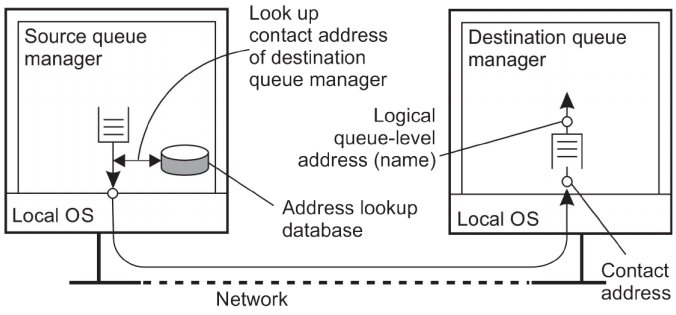

Figure 11: Queue Manager

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

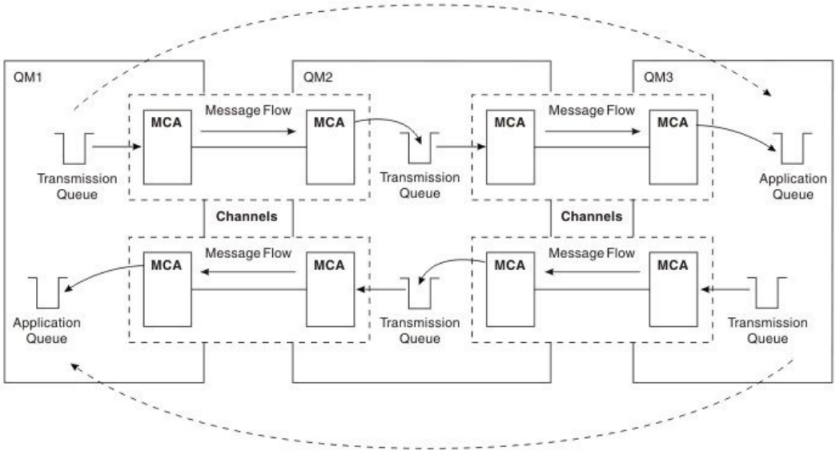

Figure 12: Overlay network

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

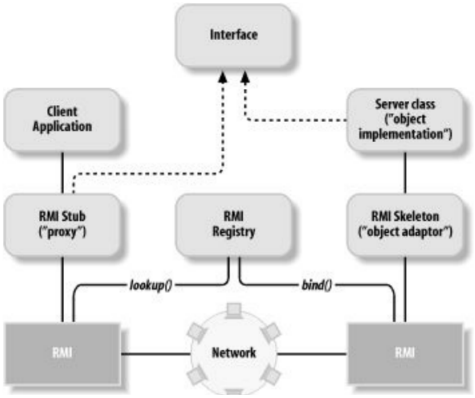

Figure 13: Architettura di RMI

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.

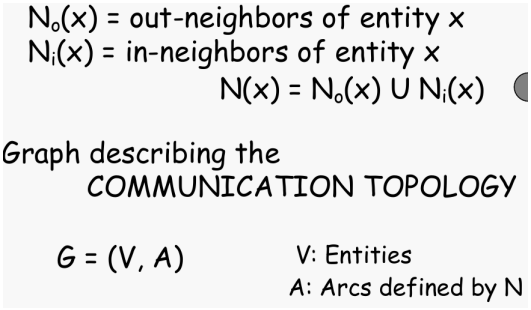

Figure 15: Come rappresentare la topologia di rete

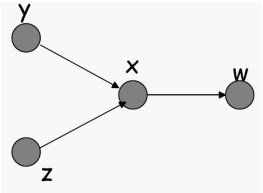

Figure 16: Topologia

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.

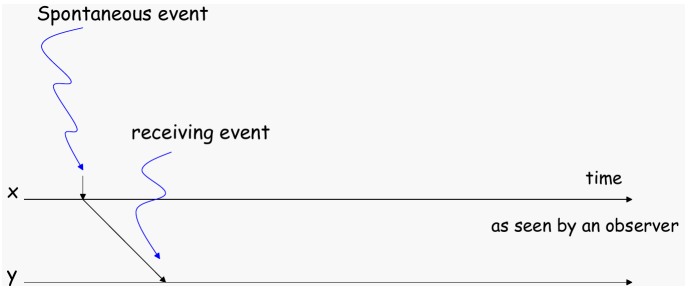

Figure 18: Stato x Evento

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**: $p \in EK_t[W]\_iff \forall x \in W \ (p \in LK_t[x])$

I **tipi di conoscenza** includono quelle **topologiche**, **metriche** (numero di nodi, diametro, eccentricit), **senso della direzione** (informazioni sui link, informazioni sulle label). Importante sottolineare come **al crescere delle conoscenze l'algoritmo diventi meno portatile**. Gli algoritmi generici non utilizzano nessuna conoscenza.

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

Assunzioni: (**BL, CN, TR**) Bidirectional link, Connectivity (ogni entit capace di raggiungere l'altra), Total reliability. + (**UI+**).
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 ter-

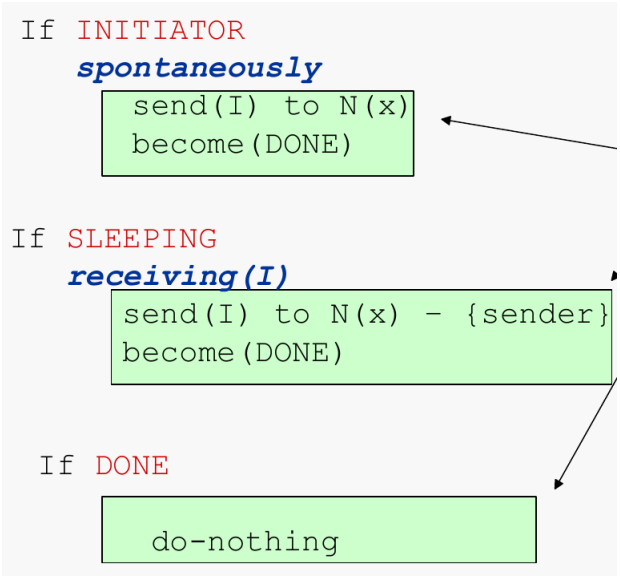

Figure 19: Algoritmo Flooding

mina 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**.
Per quanto riguarda la **complessit dei messaggi**: vengono scambiati 2 messaggi per ogni link

$$\sum_x N(x) = 2m \rightarrow 2m = O(m)$$

nello specifico:

$$|N(s)| + \sum_{x \neq s}(N(x) - 1) = \sum_x (N(x) - \sum_x 1) = 2m - (n-1)$$

Per quanto riguarda la complessit in tempo abbiamo:

$$r(s) = Max_x(d(x,s)) = eccentricity \leq Diameter(G) \leq n-1$$

con:

$$Diameter(G) = Max_x(r(x))$$

2.3 Flooding in reti con caratteristiche particolari

Un algoritmo che implementi il broadcast in un sistema distribuito varia la sua efficienza in base alla topologia di rete. Vediamo alcuni casi:

2.3.1 Broadcast in un Hypercube

Per $k = 1$ un hypercube un grafo che presenta due nodi collegati da un link. Un Hypercube H_k di dimensione $k > 1$ ottenuto prendendo due hypercube di dimensione $k-1-H_k^1-1$ e collegando i nodi con nome uguale con un link etichettato. I nuovi nomi dei nodi sono ottenuti aggiungendo il prefisso 1 o il prefisso 0 ai nomi precedenti. Le label dei link sono ottenute contando i bit di differenza tra il nome dei due nodi collegati. Le etichette sono simmetriche rispetto ai nodi connessi dal link

Ricordiamo che i nomi dei nodi sono utilizzati solo a scopo de-

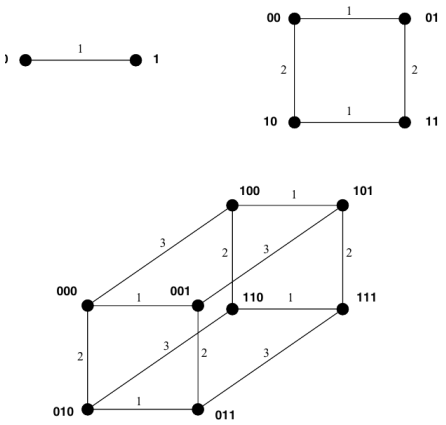

Figure 20: Hypercube

scrittivo e non sono conosciuti dalle entit. Al contrario in nomi delle etichette sono noti alle entit per l'assioma di local orientation.

Vediamo la **complessit** del flooding per questa particolare topologia. Un hypercube di dimensione k ha $n = 2^k$ **nodi**. Pertanto il **numero di link** :

$$m = nk/2 = O(nlog(n))$$

il costo del flooding pertanto:

$$2m - (n-1) = nlog(n) - (n-1) = (nlog(n)/2) + 1 = O(nlog(n))$$

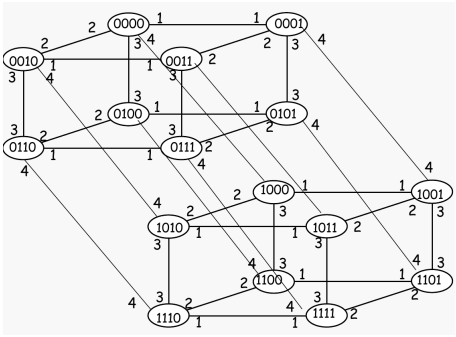

Figure 21: Costruzione hypercube

Possiamo utilizzare le propriet topologiche dell'hypercube per ottenere un broadcast ancora pi efficiente:

1. L'iniziatore invia il messaggio a tutti i suoi vicini;
2. Un nodo che riceve un mesaggio dal link l, lo invia solo ai link con etichetta $l^1 < l$

Con questa modifica il flooding coster soltanto (n-1) (messaggi).
La **correttezza** dell'algoritmo data dal seguente lemma: *per ogni paio di nodi x,y esiste un path unico di etichette decrescenti.* Il risultato che il

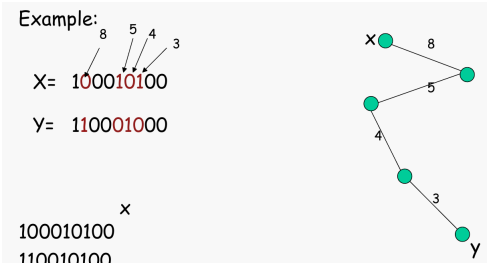

Figure 22: Lemma

messaggio crea uno **spanning tree** e ogni nodo raggiunto da tale messaggio. La complessit risulta essere quella ottimale (n-1) perch le entit ricevono l'informazione solo una volta. La complessit ideale in tempo k poich l'eccentricit di ogni nodo k.

2.3.2 Broadcast in un grafo completo

Notiamo che nel caso di un grafo completo il flooding ha complessit

$$2m - (n-1) = O(n^2)$$

con complessit per i messaggi di:

$$(n-1)$$

2.3.3 Lower bound

Torniamo al caso generale del broadcast e cerchiamo una limitazione inferiore per la sua complessit.
Teorema:
Ogni algoritmo di broadcast richiede, nel caso pessimo, O(m) messaggi.
La prova avviene per contraddizione: sia A un algoritmo che esegue broadcast in meno di m(G) messaggi. In questo caso abbiamo almeno un link in G dove nessun messaggio inviato.
VEDERE DIMOSTRAZIONE

2.4 Spanning tree construction

Per ottimizzare il costo di un algoritmo distribuito pu essere una buona soluzione quella di ottenere una topologia di rete con particolari caratteristiche. questo il caso degli **spanning tree**.
Nella teoria dei grafi uno spanning tree T di un grafo **aciclico** un suo sottografo che include tutti i vertici di G con il numero minore possibile di archi. Lo spanning tree per un grafo G **non unico**.

2.4.1 Protocollo Shout

Grazie all'assioma di **local orientation** un'entit consapevole solo delle etichette delle porte con le quali comunica con i suoi vicini. Inoltre sappiamo che i messaggi inviati ad un vicino sono prima o poi ricevuti dal destinatario (grazie all'assioma di **finite comunication delay** e la restrizione di **total reliability**). In questa configurazione iniziale un'entit ha bisogno di conoscere *solo chi tra i suoi vicini anche suo vicino nello spanning tree.* Vediamo la strategia utilizzata:

1. L'iniziatore **s** invia un messaggio ai suoi vicini "***sei tu il mio vicino?***";

2. un'entit $x \neq s$ risponde "***yes***" solo la prima volta ed in questa occasione pone a tutti i suoi vicini la stessa domanda, altrimenti risponde "***no***".

3. Ogni entit termina quando ha ricevuto una risposta da tutti i vicini.

PROTOCOL Shout

- Status: $\mathcal{S}$ = {INITIATOR,IDLE,ACTIVE,DONE};
  $\mathcal{S}_{INIT}$ = {INITIATOR,IDLE};
  $\mathcal{S}_{TERM}$ = {DONE}.
- Restrictions: **R** ;UI.

```
INITIATOR
        Spontaneously
        begin
            root:= true;
            Tree-neighbors:=∅;
            send(Q) to N(x);
            counter:=0;
            become ACTIVE;
        end

IDLE
        Receiving(Q)
        begin
            root:= false;
            parent:= sender;
            Tree-neighbors:={sender};
            send(Yes) to {sender};
            counter:=1;
            if counter=|N(x)| then
               become DONE
            else
               send(Q) to N(x) − {sender};
               become ACTIVE;
            endif
        end

ACTIVE
        Receiving(Q)
        begin
            send(No) to {sender};
        end

        Receiving(Yes)
        begin
            Tree-neighbors:=Tree-neighbors ∪{sender};
            counter:=counter+1;
            if counter=|N(x)| then become DONE; endif
        end

        Receiving(No)
        begin
            counter:=counter+1;
            if counter=|N(x)| then become DONE; endif
        end
```

Figure 23: Algoritmo protocollo shout

Osservando la struttura dell'algoritmo chiaro che risulta dalla composizione dei protocolli **flooding + reply**.

2.4.2 Correttezza

Sappiamo che flooding corretto e dunque sappiamo che ogni entit ricever Q e che per costruzione risponder *yes* o *no* ad ogni Q che riceve.
Per provare la correttezza dobbiamo provare che la sotto-rete G^1 definita da tutti gli alberi di vicini, uno spanning tree di G.
Sappiamo che:

- Se x un tree-neighbor di y allora y un tree-neighbor di x;
- Se x invia *yes* ad y, allora x un tree-neighbor di y ed connesso all'iniziatore da una catena di **yes**;
- Ogni x (a parte l'iniziatore) risponde *yes* solo una volta (quando diventa **active** risponde no ad ogni richiesta).

Poich ogni entit $x \neq y$ invia un solo *yes* allora G^1 contiene tutte le entit di G, connesso e non contiene cicli e dunque uno spanning tree di G.
Importante ricordare che Shout termina per **terminazione locale** ovvero che ogni entit conosce quando la propria esecuzione terminata (quando entra nello stato **done**). Nemmeno l'iniziatore a conoscenza della **terminazione globale** (situazione molto comune in un algoritmo distribuito).

2.4.3 Costo computazionale

Dal momento che shout definito come flood + reply studiarne la complessit risulta essere molto semplice:

$$Message(SHOUT) = 2Message(FLOOD) = 4m - 2n + 2$$

Dal momento che $O(m)$ un **lower bound** diciamo che shout **asintoticamente ottimo.** Nello specifico:

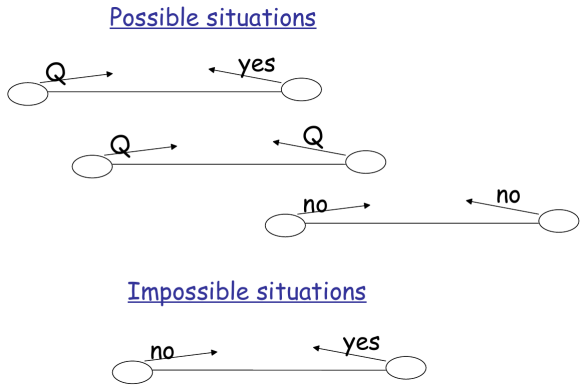

Figure 24: Shout: possibili situazioni

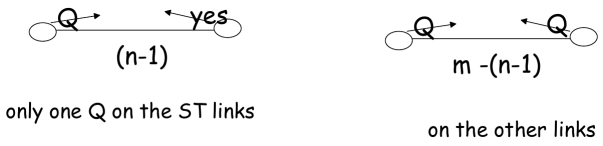

Figure 25: Totale dei messaggi Q (sei mio vicino?)

Total: $2(m-(n-1)) + (n-1)$
$= 2m - n + 1$

Figure 26: Totale dei messaggi Q: formula

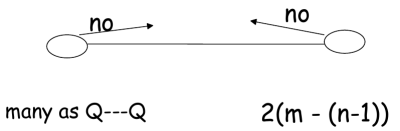

Figure 27: Totale dei messaggi no

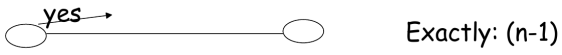

Figure 28: Totale dei messaggi yess

2.4.4 Possibili migliorie

Possiamo modificare l'algoritmo in modo tale da eliminare i messaggi ***no***. Ricordiamo che i delay di consegna dei messaggi sono assunti finiti ma sono

```
2m - n + 1 + 2(m - (n-1)) + n-1
= 2m -n +1+2m -2n +2 +n - 1
= 4m -2n + 2
```

Figure 29: Totale dei messaggi scambiati

per definizione impredicibili. Non possiamo dunque fare a meno dei messaggi di no in quanto non possiamo semplicemente attendere lo scadere dell'upper-bound di consegna dei messaggi per capire se un'entit risponder *yes* o meno. la in quanto sono utilizzati per la terminazione local Per farlo si interpretano i messaggi Q ricevuti come dei no (vedi figura). La complessit si riduce a:

$$Messages(SHOUT) = 2siomenom$$

Un'altra possibile miglioria implementata con lo scopo di ottenere **termi-**

```
receiving(Q) (to be interpreted as NO)

        counter := counter +1
        if counter = |N(x)|
                become DONE
```

Figure 30: Shout senza messaggi no

nazione globale per l'algoritmo. Si introducono degli **ACK** che permettono di notificare alla root quando terminare globalmente l'algoritmo. L'idea quella di introdurre una procedura **CHECK** che permette ad un nodo, alla ricezione di un messaggio e qualora tutti i propri vicini abbiano risposto, di controllare di essere una **foglia**: in caso affermativo la foglia invia un **ACK** al proprio parent. Il parent attende di ricevere un ACK dai tutti i figli per inviare un ACK al padre. Una volta che l'ACK giunge alla root, quest'ultima fa partire dei messaggi di **termination**. In poche parole gli ACK si muovono dalle foglie verso la radice, mentre i messaggi di termination svolgono il percorso contrario.

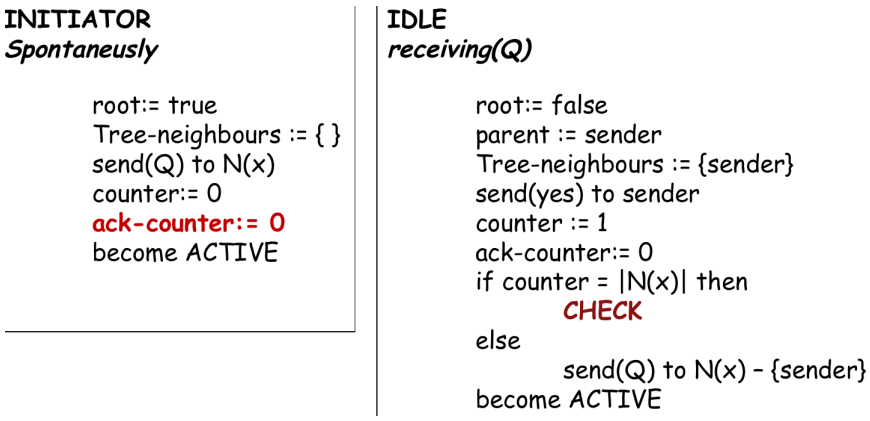

Figure 31: Algoritmo Shout con global termination 1

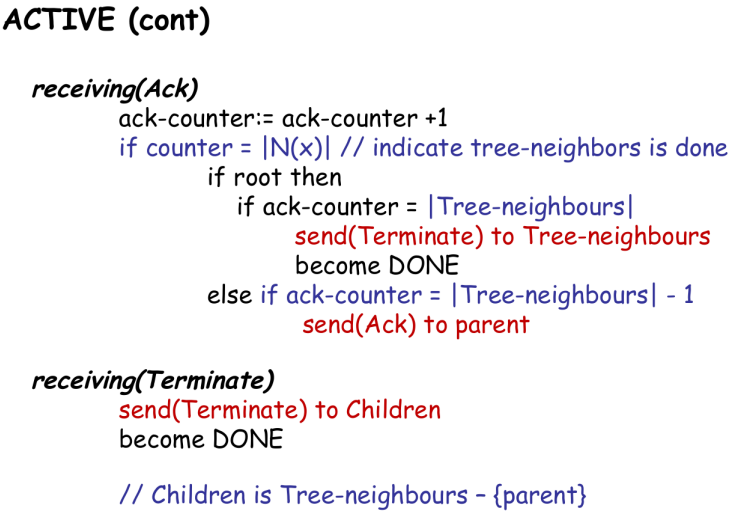

Figure 32: Algoritmo Shout con global termination 2

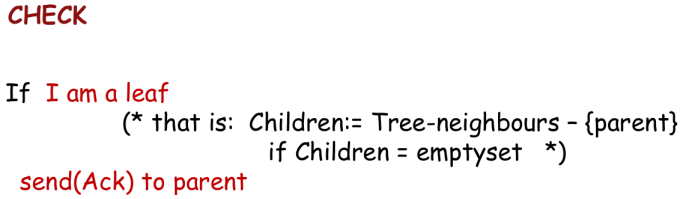

Figure 33: Algoritmo Shout con global termination

2.4.5 Iniziatore mutliplo

Abbiamo implementato l'algoritmo *shout* assumendo l'esistenza di un **iniziatore unico**. Tuttavia questa un'assunzione molto forte da considerare per un sistema distribuito. In figura si mostra cosa accade nel caso di multipli iniziatori: presi i nodi x, y, z connessi l'uno con l'altro, con x e y iniziatori, si vede facilmente che se il messaggio Q inviato da x arriva prima a z allora i link (x, y) e (y, z) non saranno presenti nello spanning tree. L'algoritmo di fatto costruisce una **spanning forest** non connessa. Questo risultato parti-

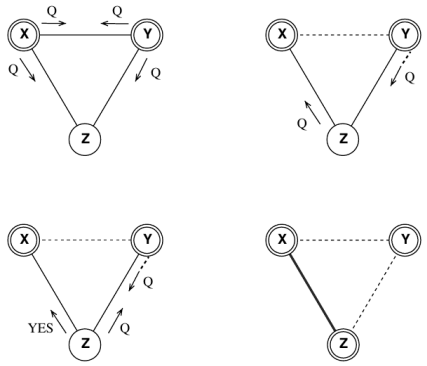

Figure 34: Iniziatori multipli in Shout

colare avvallato dal **risultato di impossibilit**:
Teorema: *il problema della costruzione di un spanning tree deterministicamente impossibile assumendo R.*
Ci significa che non esiste un protocollo deterministico che termina sempre in tempo finito. La prova viene data per assurdo: l'idea che le entit hanno lo stesso codice e perci, iniziando simultaneamente nello stesso stato, riceveranno gli stessi messaggi e svolgeranno le stesse computazioni, trovandosi sempre negli stessi stati (tutte le entit sono iniziatori). Il protocollo per essere corretto deve terminare in questa configurazione: x ha la label 2 nella lista, y ha la label 1, mentre z le ha tutte e due. L'assurdo si rileva nel fatto che le entit avrebbero valori distinti anche se gli stati e le computazioni svolte risultano le stesse.
Se vogliamo costruire uno spanning tree in un contesto distribuito abbiamo dunque bisogno di un algoritmo che esegue la **leader election**.

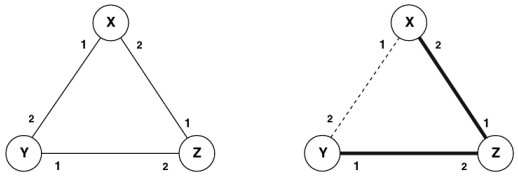

Figure 35: Prova risultato impossibilit

2.4.6 SPT: Depth First Search

Sappiamo che una visita in profondit di un grafo restituisce uno spanning tree di tale grafo. L'idea quella di utilizzare un token per identificare il nodo corrente:

- Quando un nodo viene visitato per la prima volta, tiene traccia di chi il mittente ed inoltra il token ad uno dei suoi vicini non ancora visitati.
- Quando un vicino riceve il token, se gi stato visitato marca l'arco come **back-edge** e restituisce il token, altrimenti inoltra sequenzialmente il token a tutti i suoi vicini sequenzialmente.
- Se non ci sono pi vicini non visitati ritorna il token (**reply**) al nodo dal quale per primo ha ricevuto il token.
- Una volta ricevuto un reply, inoltra il token ad un altro vicino non visitato.

States S={INITIATOR, IDLE, VISITED, DONE}
Sinit = {INITIATOR, IDLE}
Sterm = {DONE}

INITIATOR
Spontaneusly

Unvisited := N(x)
initiator := true
VISIT

IDLE
receiving(T) // T token message

entry := sender
Unvisited := N(x) - sender
initiator := false
VISIT

Figure 36: Algoritmo DFS 1

FARE IMMAGINE CON SLIDE DA 30 a 41

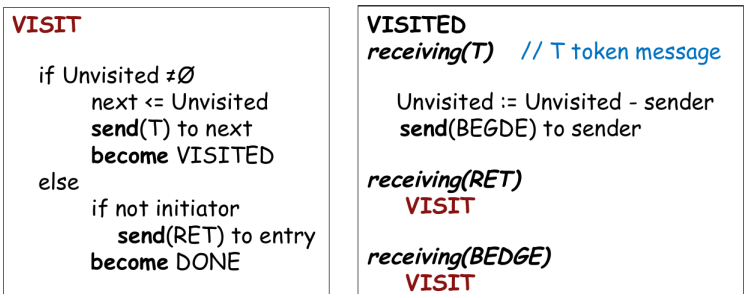

Figure 37: Algoritmo DFS 2

Per quanto riguarda la **complessit in messaggi** abbiamo che i messaggi per link sono 2 (il destinatario del token risponde return se visitato la prima volta o back se gi visitato). Dunque:

$$2m = O(m)$$

con

$$O(m) = lowerbound$$

Allo stesso modo la **complessit in tempo** risulta essere:

$$2m = O(m)$$

con

$$O(n) = lowerbound$$

2.4.7 DF: migliorie

Una prima idea quella di eliminare i messaggi **back** che costituiscono la maggioranza del totale dei messaggi inviati. Per farlo utilizziamo **notification** e i messaggi di **ACK**.

- L'idea che il nodo corrente informa i suoi vicini di essere stato visitato.
- I vicini rispondono con messaggi di ACK.
- Il mittente dunque segna tutti gli archi dal quale riceve risposta come back-edge.
- Sceglie un vicino da visitare e marca quel link come appartenente allo spanning tree.

INSERIRE IMMAGINI SLIDE 44-48

Per quanto riguarda la **complessit in messaggi** abbiamo che ogni entit riceve un **token** ed invia un **return**

$$2(n-1)$$

Inoltre ogni entit invia 1 **visited** a tutti i vicini tranne al mittente (lo stesso vale per i messaggi di ACK).

$$\sum |N(s)| + \sum_{x \neq s} (|N(x)| - 1)$$

Il totale dunque $4m$ Per quanto riguarda la **complessit in tempo** abbiamo che i l'invio dei Token e il Return sono inviati sequenzialmente con complessit $2(n-1)$ mentre l'invio degli ack e dei messaggi visited svolto in parallelo con complessit $2n$. Il totale risulta

$$4n - 2$$

FINIRE DF++

3 Computazione negli alberi

In questa sezione analizzato il contesto delle computazioni distribuite in una topologia in forma di **albero**. Anche in questo caso valgono le restrizioni standard definite da **R**, in aggiunta ogni nodo sa se una **foglia** o un **nodo interno** (se ha un solo vicino o pi di uno).

3.1 Saturation

La **Saturazione** una tecnica base che pu essere utilizzata come strumento di partenza per eseguire computazioni in un sistema distribuito. Il protocollo si declina in 3 parti: **Attivazione**, **Saturazione**, **Risoluzione**. La fase di risoluzione dipender dall'applicazione specifica del protocollo (definisce cosa facciamo con i messaggi di saturation ricevuti) anche se solitamente viene utilizzata come fase di notifica per tutte le entit (con lo scopo di ottenere *local termination*). Un protocollo "troncato" come questo chiamato **plug-in**

(un protocollo nel quale non tutte le entit entrano in stato terminale). Per far si che diventi un protocollo necessario definire ulteriori azioni da svolgere. ***Il protocollo pu essere avviato da un qualsiasi numero di initiator.*** Vediamo le fasi in dettaglio:

1. **Attivazione**: questa fase un semplice ***wake-up***. Ogni initiator invia un messaggio di wake-up a tutti i suoi vicini e diventa *active*. Ogni non initiator che riceve il messaggio diviene active a sua volta e inoltra il messaggio ai suoi vicini (escluso il mittente del messaggio)). I nodi gi attivi ignorano altri messaggi di wake up. In **tempo finito** tutti i nodi divengono attivi, incluse le foglie (per le assunzione di **total reliability** e l'assioma di **finite comunication delay**)

2. **Saturazione**: questa fase inizializzata dalle foglie che inviano il messaggio **M** di saturazione al loro unico vicino (il parent), entrando cos nello stato di ***processing***. Invece, ogni nodo intermedio attende di ricevere un messaggio di saturation da tutti i suoi vicini meno uno. All'ultimo vicino rimasto ed identificato dunque come il parent, il nodo intermedio invia un messaggio di saturation (entrando nello stato *processing*). Se un nodo nello stato di processing riceve un messaggio dal parente allora entra nello stato ***saturated***.

3. **Risoluzione**: dipende dall'applicazione.

```
S = {AVAILABLE, ACTIVE, PROCESSING, SATURATED}
Sinit = AVAILABLE
Restrictions: R;T

AVAILABLE    I haven't been activated yet
Spontaneously
                send(Activate) to N(x);
                Initialize;
                Neighbours:= N(x)
                if |Neighbours|= 1 then
                        Prepare_Message; // M := "Saturation"
/* special case if
I am a leaf */          parent « Neighbours;
                        send(M) to parent;
                        become PROCESSING;
                else
                        become ACTIVE;
```

Figure 38: Algoritmo Saturazione 1

```
AVAILABLE
Receiving(Activate)
                send(Activate) to N(x) - {sender};
                Initialize;
                Neighbours:= N(x);
                if |Neighbours|= 1  then
                        Prepare_Message;  //M := "Saturation"
                        parent « Neighbours;
/* special case if
I am a leaf  */          send(M)  to  parent;
                        become PROCESSING;
                else
                        become ACTIVE;
```

Figure 39: Algoritmo Saturazione 2

```
ACTIVE                  I haven't started the saturation
Receiving(M)            phase yet
                Process_Message;
                Neighbours:= Neighbours - {sender};
                if |Neighbours| = 1 then
                        Prepare_Message;  //M := "Saturation"
                        parent « Neighbours;
                        send(M) to parent;
                        become PROCESSING;
```

```
PROCESSING              I have already started the
                        saturation phase
receiving(M)
        Process_Message;
        become SATURATED; // Resolve
```

Figure 40: Algoritmo Saturazione 3

3.1.1 Prova di correttezza

Lemma: *Esattamente due nodi in stato di processing diventeranno saturati, inoltre questi nodi sono vicini ma anche l'uno il parent dell'altro*

Prova: dzal codice sappiamo che un nodo invia il messaggio M solo al suo parente e diviene saturato solo se riceve un messaggio M dal parent. Scegliendo arbitrariamente un nodo della rete x_i, se attraversiamo gli up-edges (archi che collegano x_i al proprio parent) prima o poi troviamo un nodo saturato s_1 (questo perch non ci sono loop nel grafo). Il nodo s_1 diventato saturato perch ha ricevuto un messaggio da un nodo s_2 mentre era nello stato di processing (dunque ha gi ricevuto M da un altro nodo quando era active). Dal punto di vista di s_2 questo significa che egli nello stato processing e che considera s_1 il suo parent. Dunque, quando s_2 riceve un messaggio da s_1 questo diventa saturato a sua volta (i due nodi sono l'uno il parent dell'altro). Considerando il caso in cui i nodi saturati sono pi di due allora

significherebbe che esistono due nodi x,y saturati per i quali $d(x, y) \leq 2$. Tuttavia se consideriamo un nodo z tra i due nodi x,y vediamo che z non pu inviare il messaggio M ad entrambi i nodi e dunque uno dei nodi non pu essere saturato.

Importante: impredicibile quale coppia di nodi divenga saturata, questo per via dei delay di consegna dei messaggi (ovviamente incide anche la scelta degli iniziatori).

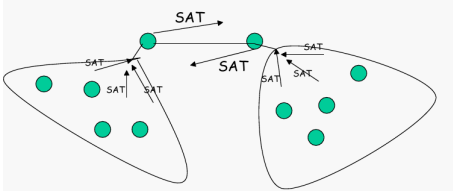

Figure 41: Algoritmo Saturazione: prova

3.1.2 Complessit

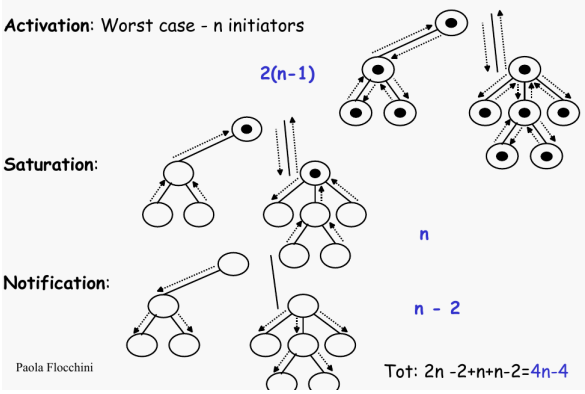

Figure 42: Algoritmo Saturazione: complessit caso peggiore

3.1.3 Ricerca del minimo con saturazione

In questa sezione vediamo come si pu implementare la fase di risoluzione del protocollo per la ricerca del minimo. In questa configurazione ogni nodo

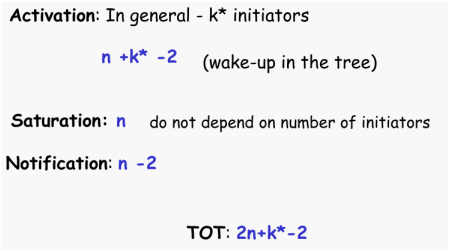

Figure 43: Algoritmo Saturazione: complessit caso generale con n iniziatori

possiede un valore $v(x)$, al termine dell'algoritmo ogni nodo consapevole di possedere il valore minimo ed entra nello stato appropriato (*minimum* o *large*). Pi entit possono avere il valore minimo.
Il problema pu essere risolto, nel caso di un rooted tree (esiste un nodo speciale che la radice e abbiamo orientamento degli archi) eseguendo **convergecast**: a partire dalle foglie i nodi determinano il valore minimo e lo inviano verso la radice. Il minimo dunque individuato dalla radice che si occupa di comunicarlo in broadcast agli altri nodi. ***Assumere l'esitenza di una radice un'assunzione molto forte: equivale ad assumere l'esistenza di un leader all'interno della topologia.***
Per questo motivo usiamo **saturation** per risolvere il problema senza bisogno di queste informazioni. L'unica modifica effettuata riguarda la fase di processing:

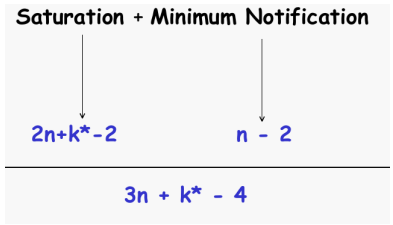

Figure 44: Ricerca minimo con Saturazione: complessit

3.1.4 Computazione distribuita di funzioni

Il problema vede la computazione di una funzione all'interno di un sistema nel quale i suoi argomenti sono distribuiti nei nodi della topologia.
Assumiamo la funzione f essere *associativa* e *commutativa*. Questo tipo di

funzioni, assieme ai suoi parametri, sono dette **Semigruppi commutativi**.

Semigroup: An algebraic structure consisting of a set **S** and a binary operator **#** which is **associative**

$$a \# (b \# c) = (a \# b) \# c$$

Commutative semigroup: A semigroup where F is also commutative

$$a \# b = b \# a$$

Figure 45: Semigruppi commutativi

```
PROCESSING
receiving(Notification)
        send(Notification)  to N(x) - parent
        result := Received_Value;
        become DONE;
```

```
Initialize
        if v(x) is not nil then
            result := f(v(x))
        else
            result := nil

Prepare_Message
        M := ("Saturation", result);
```

Figure 46: Algoritmo computazione distribuita di funzioni 1

```
Process_Message
        if Received_value is not nil then
            if result is not nil then
                result := f(result, Received_value)
            else
                result := f(Received_value)


Resolve
        Notification := ("Resolution", result);
        send(Notification) to N(x) - parent;
        become DONE;
```

Figure 47: Algoritmo computazione distribuita di funzioni 2

```
Process_Message
        if Received_value is not nil then
            if result is not nil then
                result := f(result, Received_value)
            else
                result := f(Received_value)


Resolve
        Notification := ("Resolution", result);
        send(Notification) to N(x) - parent;
        become DONE;
```

Figure 48: Computazione distribuita di funzioni: complessit