

# DPIoT - Riassunto

Tommaso Puccetti

Studente presso Universita degli studi di Firenze

February 4, 2020

## Contents

<b>1</b>	<b>Introduzione ai sistemi distribuiti</b>	<b>8</b>
1.1	Intro . . . . .	8
1.1.1	Obiettivi di un sistema distribuito . . . . .	9
1.1.2	Nascondere le latenze di comunicazione . . . . .	12
1.1.3	Java socket . . . . .	13
<b>2</b>	<b>Sistemi Distribuiti</b>	<b>14</b>
2.1	Architetture . . . . .	14
2.1.1	Architetture software: layered architecture . . . . .	15
2.1.2	Architetture software: Object-based architecture . . . . .	15
2.1.3	Architetture software: Service oriented . . . . .	16
2.1.4	Restful architecture . . . . .	17
2.1.5	Publish subscribe architecture . . . . .	18
2.1.6	Tuple space architecture . . . . .	19
2.1.7	Architetture di sistema: client/server . . . . .	19
2.1.8	Architettura peer to peer . . . . .	21
2.1.9	P2P overlay network non strutturata . . . . .	21
2.1.10	P2P overlay network strutturata . . . . .	22
2.2	Jersey . . . . .	22
<b>3</b>	<b>Communication Mechanisms</b>	<b>24</b>
3.1	Middleware . . . . .	24
3.2	Coordinazione diretta . . . . .	26
3.3	Remote Procedure Call . . . . .	26

3.3.1	Passaggio di parametri . . . . .	28
3.3.2	Implementare RPC . . . . .	29
3.3.3	RPC Asincrono . . . . .	30
3.3.4	Binding . . . . .	31
3.4	Message Oriented Middleware . . . . .	31
3.4.1	Queue Manager . . . . .	34
3.4.2	Eterogeneit: Message Brokers . . . . .	35
3.5	Java RMI . . . . .	35
3.6	gRPC . . . . .	37
<b>4</b>	<b>Basic distributed algorithms</b>	<b>38</b>
4.1	Contesto . . . . .	38
4.1.1	Assiomi . . . . .	40
4.1.2	Restrizioni . . . . .	40
4.1.3	Tempo ed Eventi . . . . .	41
4.1.4	Livelli di Conoscenza . . . . .	42
4.2	Broadcast . . . . .	43
4.2.1	Flooding . . . . .	43
4.3	Flooding in reti con caratteristiche particolari . . . . .	44
4.3.1	Broadcast in un Hypercube . . . . .	44
4.3.2	Broadcast in un grafo completo . . . . .	47
4.3.3	Lower bound . . . . .	47
4.4	Spanning tree construction . . . . .	48
4.4.1	Protocollo Shout . . . . .	48
4.4.2	Correttezza . . . . .	50
4.4.3	Costo computazionale . . . . .	50
4.4.4	Possibili migliorie . . . . .	51
4.4.5	Iniziatore mutliplo . . . . .	54
4.4.6	SPT: Depth First Search . . . . .	55
4.4.7	DF+ . . . . .	57
4.4.8	DF++ . . . . .	58
4.4.9	Smart traversal . . . . .	59
4.4.10	Wake-up . . . . .	60
4.5	Computazione negli alberi . . . . .	61
4.5.1	Saturation . . . . .	61
4.5.2	Prova di correttezza . . . . .	62
4.5.3	Complessit . . . . .	64
4.5.4	Ricerca del minimo con saturazione . . . . .	64

4.5.5	Computazione distribuita di funzioni . . . . .	65
4.6	Leader Election . . . . .	66
4.6.1	Risultato di impossibilit . . . . .	67
4.6.2	Election negli alberi . . . . .	68
4.6.3	Leader election in un anello: All the Way . . . . .	69
4.6.4	All the way: correttezza e terminazione . . . . .	70
4.6.5	AsFar (as it can) . . . . .	72
4.6.6	AsFar: terminazione . . . . .	72
4.6.7	AsFar: complessit . . . . .	72
4.6.8	Controlled Distances . . . . .	75
4.6.9	Controlled Distances: correttezza e costo . . . . .	76
4.6.10	Stage . . . . .	78
4.6.11	Stage: correttezza e complessit . . . . .	79
4.6.12	Stages: rimuovere message ordering . . . . .	80
4.6.13	Stages: correttezza terminazione . . . . .	81
4.6.14	Alternating Steps . . . . .	82
4.6.15	Unidirectional o Bidirectional links? . . . . .	85
4.7	Yo-Yo: algoritmo distribuito per la ricerca del minimo . . . . .	86
4.7.1	Struttura . . . . .	86
4.7.2	Terminazione . . . . .	88
4.8	Election in una rete dinamica . . . . .	90
4.8.1	The Bully Algorithm: introduzione . . . . .	90
4.8.2	Bully: algoritmo . . . . .	91
4.8.3	Elezione in una rete wireless Ad-Hoc . . . . .	92
4.8.4	Rete Ad Hoc: algoritmo leader election . . . . .	94
4.9	Sincronizzazione . . . . .	97
4.9.1	Introduzione . . . . .	97
4.9.2	Sincronizzazione di clock fisici . . . . .	98
4.9.3	Network Time Protocol (NTP) . . . . .	100
4.9.4	Clock logici . . . . .	101
4.9.5	Vector Clocks . . . . .	103
4.9.6	Mutua esclusione distribuita . . . . .	104
4.9.7	Centralized Algorithm . . . . .	105
4.9.8	Traversal Based Algorithm . . . . .	106
4.9.9	AskAll Algorithm . . . . .	106
4.9.10	Quorum-based algorithm . . . . .	107
4.10	JBotSim . . . . .	107

<b>5 IOT: Protocolli livello applicativo</b>	<b>108</b>
5.1 Introduzione . . . . .	108
5.1.1 Mobile Ad-Hoc Network . . . . .	108
5.1.2 WSN: reti di sensori . . . . .	109
5.1.3 Internet Of The Things . . . . .	113
5.1.4 Problemi rilevanti in IOT . . . . .	113
5.2 MQTT . . . . .	113
5.2.1 Introduzione MQTT . . . . .	113
5.2.2 Paradigma Produttore/Consumatore . . . . .	115
5.2.3 Produttore/consumatore in MQTT . . . . .	117
5.2.4 QOS in MQTT . . . . .	119
5.3 COAP . . . . .	122
5.3.1 Introduction . . . . .	122
5.3.2 Coap . . . . .	122

## List of Tables

## List of Figures

1 Overlay network . . . . .	9
2 Esempio di interfacci (IDL CORBA) . . . . .	11
3 Tipi di scalabilit . . . . .	12
4 Tipi di scalabilit . . . . .	12
5 Asincrono vs sincrono . . . . .	13
6 Spostare la computazione sul client . . . . .	13
7 Altre sfide nella modellazione di un sistema distribuito . . . . .	14
8 Metodi java socket . . . . .	14
9 Passi per creare un server con java socket . . . . .	14
10 Layered architecture . . . . .	15
11 Layered architecture . . . . .	16
12 Layered architecture . . . . .	16
13 Layered architecture . . . . .	17
14 Risorse rest . . . . .	18
15 Operazioni rest . . . . .	18
16 Tipi di messaggi . . . . .	19
17 Architettura client server . . . . .	19

18	two tier . . . . .	20
19	three tier . . . . .	20
20	p2p architecture . . . . .	21
21	p2p architecture . . . . .	22
22	Esempio invocazione . . . . .	23
23	@path . . . . .	23
24	@path . . . . .	23
25	Annotazioni . . . . .	24
26	Annotazioni . . . . .	24
27	Livello Middleware . . . . .	25
28	Chiamata a procedura locale vs remota . . . . .	27
29	Funzionamento RPC . . . . .	27
30	Xml . . . . .	29
31	Marshaling in Java . . . . .	29
32	Oggetti remoti e locali . . . . .	30
33	RPC tradizionale e asincrona . . . . .	31
34	Callback . . . . .	31
35	Binding . . . . .	32
36	Code . . . . .	32
37	Queue Manager . . . . .	34
38	Overlay network . . . . .	34
39	Architettura di RMI . . . . .	35
40	Definizione di un interfaccia con gRPC IDL . . . . .	38
41	Client in grpc . . . . .	38
42	Come rappresentare la topologia di rete . . . . .	40
43	Topologia . . . . .	40
44	Labels . . . . .	40
45	Stato x Evento . . . . .	42
46	Algoritmo Flooding . . . . .	43
47	Hypercube . . . . .	45
48	Costruzione hypercube . . . . .	45
49	Lemma . . . . .	46
50	Complessità floodin in reti particolari . . . . .	47
51	Dimostrazione lower bound broadcast . . . . .	48
52	Algoritmo protocollo shout . . . . .	49
53	Shout: possibili situazioni . . . . .	51
54	Totale dei messaggi Q (sei mio vicino?) . . . . .	51
55	Totale dei messaggi Q: formula . . . . .	51

56	Totale dei messaggi no . . . . .	51
57	Totale dei messaggi yess . . . . .	51
58	Totale dei messaggi scambiati . . . . .	52
59	Shout senza messaggi no . . . . .	52
60	Algoritmo Shout con global termination 1 . . . . .	53
61	Algoritmo Shout con global termination 2 . . . . .	53
62	Algoritmo Shout con global termination . . . . .	53
63	Iniziatori multipli in Shout . . . . .	54
64	Prova risultato impossibilit . . . . .	55
65	Algoritmo DFS 1 . . . . .	56
66	Algoritmo DFS 2 . . . . .	56
67	Algoritmo DF++ . . . . .	58
68	Complessit DF a confronto . . . . .	59
69	Complessit DF a confronto . . . . .	60
70	Complessit wake up in reti speciali . . . . .	61
71	Algoritmo Saturazione 1 . . . . .	62
72	Algoritmo Saturazione 2 . . . . .	63
73	Algoritmo Saturazione 3 . . . . .	63
74	Algoritmo Saturazione: prova . . . . .	64
75	Algoritmo Saturazione: complessit caso peggiore . . . . .	65
76	Algoritmo Saturazione: complessit caso generale con n iniziatori . . . . .	65
77	Ricerca minimo con Saturazione: complessit . . . . .	66
78	Semigruppi commutativi . . . . .	66
79	Algoritmo computazione distribuita di funzioni 1 . . . . .	66
80	Algoritmo computazione distribuita di funzioni 2 . . . . .	67
81	Computazione distribuita di funzioni: complessit . . . . .	67
82	Root Election . . . . .	69
83	Bit complexity . . . . .	69
84	All the way . . . . .	70
85	All the way 1 . . . . .	71
86	All the way 2 . . . . .	71
87	All the way: complessit . . . . .	72
88	AsFar . . . . .	73
89	AsFar: algoritmo 1 . . . . .	73
90	AsFar. algoritmo 2 . . . . .	74
91	AsFar: complessit caso pessimo . . . . .	74
92	AsFar: complessit caso pessimo . . . . .	75
93	AsFar: complessit caso migliore . . . . .	75

94	Distanza controllata . . . . .	76
95	Distanza controllata 1 . . . . .	77
96	Distanza controllata 2 . . . . .	77
97	Distanza controllata 3 . . . . .	78
98	Algoritmo stages . . . . .	80
99	Control, sopravvisuti ad ogni stage . . . . .	82
100	Alternating steps . . . . .	83
101	Alternating steps algoritmo 1 . . . . .	83
102	Alternating steps algoritmo 2 . . . . .	84
103	Alternating steps complessit . . . . .	84
104	Alternating steps complessit 2 . . . . .	85
105	Alternating steps complessit 2 . . . . .	86
106	DAG: nozioni base . . . . .	88
107	DAG ottenuto con la fase di setup . . . . .	88
108	Dalle source ai sink . . . . .	89
109	Dai sink alle sources . . . . .	89
110	Regole di pruning . . . . .	90
111	YO-YO complessit . . . . .	90
112	Bully esempio . . . . .	92
113	Bully complessit . . . . .	93
114	Leader election per reti ad hoc . . . . .	95
115	Computazioni concorrenti 1 . . . . .	96
116	Computazioni concorrenti 2 . . . . .	96
117	Esempio timestamp erronei 1 . . . . .	97
118	Esempio timestamp erronei 2 . . . . .	98
119	Problemi di sincronizzazione in contesto distribuito . . . . .	99
120	Algoritmo di Christian . . . . .	99
121	Algoritmo di Christian . . . . .	100
122	Algoritmo di Christian . . . . .	100
123	Algoritmo di Christian . . . . .	101
124	Relazione logica tra due eventi . . . . .	101
125	Relazione logica tra due eventi . . . . .	102
126	Posizionamento orologi di lamport . . . . .	103
127	Problema logical clock . . . . .	104
128	Algoritmo centralizzato . . . . .	105
129	Algoritmo centralizzato:complessit . . . . .	106
130	Traversal based: complessit . . . . .	106
131	Complessit AskAll . . . . .	107

132	Complessit quorum based . . . . .	108
133	rete ad hoc . . . . .	110
134	Stack protocolli per rete ad hoc . . . . .	110
135	Configurazione tipica per una rete WSN . . . . .	111
136	Tecnologie per WSN . . . . .	112
137	Standard a confronto . . . . .	112
138	Standard a confronto . . . . .	113
139	Stack di rete . . . . .	114
140	Requirements IOT . . . . .	114
141	Stack di rete . . . . .	116
142	Stack con MQTT . . . . .	117
143	MQTT: qos 0 . . . . .	120
144	MQTT: qos 1 . . . . .	120
145	MQTT qos 2 . . . . .	120
146	MQTT:keepalive . . . . .	121
147	MQTT:struttura pacchetti . . . . .	121
148	Pila COAP . . . . .	122
149	Dal web ad un constrained environment . . . . .	123

# 1 Introduzione ai sistemi distribuiti

## 1.1 Intro

*”Un sistema distribuito una collezione di elementi computazionali autonomi che appaiono all’utente come un unico sistema coerente”* Che cosa sono questi elementi che possiedono potenza computazionale?

- Processi software;
- Dispositivi Hardware.

I nodi comunicano tra di loro attraverso lo **scambio di messaggi** e sono programmati per **raggiungere un obiettivo comune**. Inoltre ogni nodo autonomo e possiede la sua nozione di tempo (clock drifting). Questi nodi sono solitamente organizzati in un **Overlay network** (un grafo): ogni nodo comunica solo con i suoi vicini (l’insieme dei vicini pu essere **dinamico** o **statico**.) Un rete distribuita pu essere **aperta** o **chiusa** (richiede meccanismi di gestione degli accessi.)

Un overlay network è un grafo **connesso, costruito sopra un'altra rete** (in questo caso la rete sottostante quella fisica). Ne esistono due tipi:

- **Strutturata:** albero, anello, hypercube.
- **Non strutturata:** rete casuale.

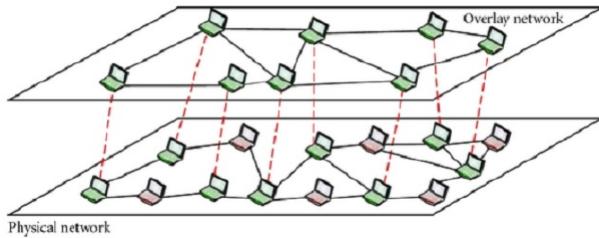


Figure 1: Overlay network

La struttura del sistema è trasparente all'utente:

- Un utente non sa dire dove la computazione ha avuto luogo;
- La posizione dei dati è indifferente.

Risulta molto difficile ottenere davvero questo tipo di trasparenza nei confronti dell'utente: i sistemi distribuiti sono complessi e composti da molti elementi che possono fallire in qualsiasi momento. Vediamo quali sono le tipologie di trasparenza che si possono ottenere:

### 1.1.1 Obiettivi di un sistema distribuito

**Il primo obiettivo** che trattiamo è la condivisione delle risorse: deve essere facile per l'utente accedere alle risorse remote che si tratti di dispositivi, potenza computazionale, files o dati.

**Il secondo** riguarda la **trasparenza** ovvero dare all'utente un'interfaccia uniforme che nasconde la natura distribuita del sistema in modo tale che quest'ultimo sia percepito come un singolo computer system. Esistono diversi tipi di trasparenza:

- **Access transparency:** ha a che fare con diverse rappresentazioni dei dati nel sistema distribuito. L'obiettivo trovare un accordo su come i dati sono presentati in modo tale che questi possano essere acceduti da sistemi diversi (differenti OS, JSON una soluzione).
- **Location transparency:** nascondere all'utente dove la risorsa richiesta collocata fisicamente. Si utilizzano **nomi logici**, indipendenti dalla locazione della risorsa (es URL).
- **Migration transparency:** nascondere il fatto che una risorsa pu essere spostata fisicamente (fare in modo che la modalit di accesso non cambi in relazione alla posizione fisica della risorsa);
- **Relocation transparency:** nasconde il fatto che una risorsa pu essere spostata fisicamente in un altra posizione del sistema **mentre utilizzata**.
- **Replication transparency:** replicare le risorse su pi macchine fondamentale in un sistema distribuito. La relativa trasparenza volta a nascondere il fatto che esistano diverse copie della risorsa nel sistema. Sono necessari **nomi logici** e **trasparenza locale** (**problema di consistenza tra copie**).
- **Concurrency transparency:** nascondere pi utenti possano accedere alla stessa risorsa lasciandola, allo stesso tempo, in uno stato consistente.
- **Failure transparency:** nascondere il fallimento di un nodo del sistema (es reindirizzando l'utente su un altro server).

Ottener la trasparenza difficile: le comunicazioni sono affette dal problema del delay di consegna dei messaggi, i nodi possono fallire e determinarne il fallimento un processo lento. Inoltre dal punto di vista delle performance conservare pi copie dei dati pu essere molto costoso. **Dobbiamo trovare un compromesso in relazione ai requirements del sistema.**

Il **terzo obiettivo l'openess**. Un sistema **aperto** se offre componenti che possono cooperare ed essere integrate in altri sistemi e se nuove risorse possono essere dinamicamente aggiunte al sistema. Le risorse sono offerte tramite i protocolli standard. Requisiti della openness:

- **Standard interfaces:** definisce la sintassi e la semantica dei servizi che ogni componente offre.
- **Interoperabilit:** due componenti forniti da due produttori diversi possono cooperare affidandosi semplicemente alle loro interfacce.
- **Portabilit:** un componente sviluppato per un sistema A può essere eseguito su un sistema differente.
- **Estendibilit:** facile aggiungere nuovi componenti o sostituirne di vecchi senza influire sul funzionamento del sistema.

Per **definire le interfacce** si utilizzano gli **IDL: interface definition language** (definire nomi di funzioni, tipo dei parametri, eccezioni possibili) **Un sistema aperto anche flessibile:** per esserlo deve essere costituito da

```
module Bank
{
    typedef float CashAmount; // Type for cash
    typedef string AccountId; // Type for account ids

    interface Account
    {
        readonly attribute AccountId account_id;
        readonly attribute CashAmount balance;

        void withdraw(in CashAmount amount) raises (InsufficientFunds);

        void deposit(in CashAmount amount);
    }
}
```

Figure 2: Esempio di interfacce (IDL CORBA)

componenti piccole e facilmente sostituibili. I sistemi tradizionali **monolitici** tendono a non esserlo. Un metodo efficace per ottenere flessibilità quello di **dividere le politiche dai meccanismi**:

- le **politiche** specificano cosa deve essere fatto;
- i **meccanismi** come deve essere fatto.

L'idea quella di modificare una politica con lo scopo di rendere più efficiente il sistema, senza intaccare i meccanismi che implementano quella politica (cambiare la priorità di schedulazione dei processi piuttosto che l'algoritmo di scheduling)

**Il quarto obiettivo la scalabilità:** la capacità del sistema di crescere e gestire carichi di lavoro sempre maggiori (tipologie in figura).

**Size scalability:** numbers of users and/or nodes/processors/resources  
**Administrative scalability:** numbers of organization or administrative domains sharing a single distributed system  
**Load scalability:** ability to expand and contract resource pools to accommodate heavier or lighter workloads  
**Functional scalability:** ability to support new functionality at minimal efforts  
**Geographic scalability:** ability to maintain performance and usability regardless of the distance between nodes

Figure 3: Tipi di scalabilit

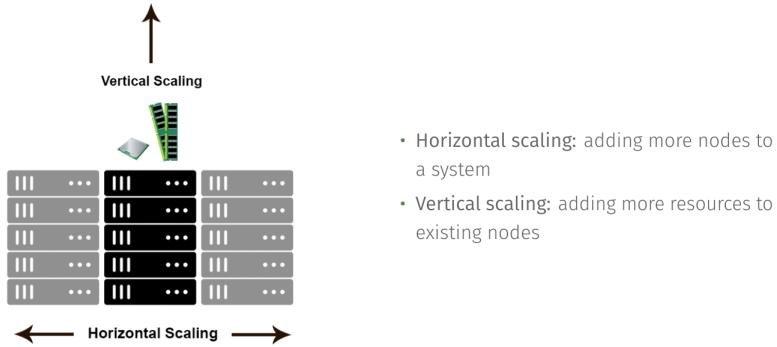


Figure 4: Tipi di scalabilit

### 1.1.2 Nascondere le latenze di comunicazione

Nascondere quanto più possibile la latenza di comunicazione fondamentale in un sistema distribuito. Nel concreto vogliamo che l'utente non attenda nell'utilizzo di un servizio remoto. In generale esistono due tipologie di comunicazione:

- **Sincrone:** il mittente interrompe l'esecuzione finché il destinatario non riceve il messaggio;
- **Asincrone:** il mittente invia il messaggio e continua con la propria esecuzione.

Un'idea per risolvere il problema potrebbe essere quella di **spostare la computazione sul cliente** (vedi figura) **Il caching** un'altra possibile soluzione (avere più copie di una risorsa posizionate in più componenti del sistema). In istanza finale, quello che vogliamo ottenere sono:

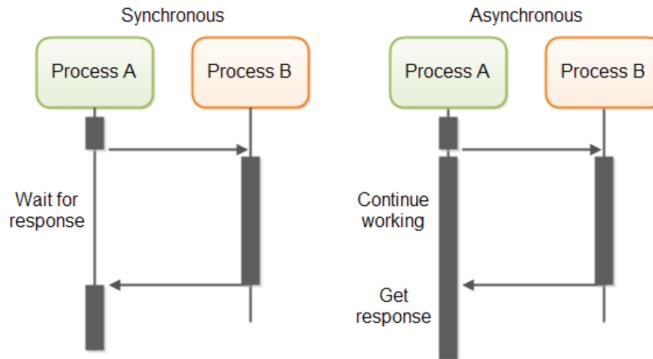


Figure 5: Asincrono vs sincrono

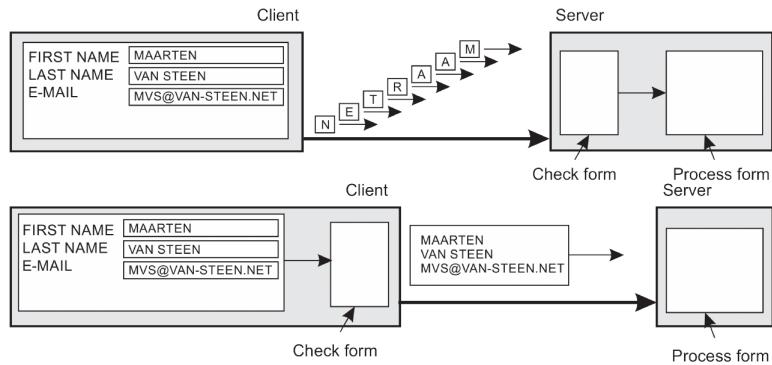


Figure 6: Spostare la computazione sul client

- **Availability:** ogni richiesta fatta il sistema si deve tradurre in una risposta. Ci implica una pronta reazione ai fallimenti e la capacità di ripristinare il sistema in seguito ad uno di essi.
- **Modularity:** il sistema deve essere composto da piccole parti.

### 1.1.3 Java socket

Le socket sono **endpoint** di un canale di comunicazione bidirezionale tra due host:

- un endpoint una coppia indirizzo IP e numero di porta.

## Other Challenges

- Security:
  - Component authentication
  - Isolate misbehaving components
  - DoS attacks
  - Information flow
- Implementation challenges:
  - What's the bottleneck?
  - How to reduce the load on bottleneck resources?
- Quality of service:
  - Throughput
  - Responsiveness
  - Load balancing

Figure 7: Altre sfide nella modellazione di un sistema distribuito

- una connessione TCP identificata da due endpoint

```
Socket() Create an unconnected socket
Socket(InetAddress address, int port) Create a socket and connects it to the
                                         specified IP address and port
Socket(String host, int port) Creates a socket and connects it to the specified
                                         port number of the named host
```

Figure 8: Metodi java socket

1. Create an instance of `ServerSocket` class binding it on a specific port
2. Listen for incoming connections on that port using its (blocking) `accept()` method  
When a client connects the `accept()` returns a `Socket` object connecting the client and the server
3. Get input and output streams to communicate with the client using  
`getInputStream()` and `getOutputStream()` methods of class `Socket`
4. Interact with the client
5. Close the connection

Figure 9: Passi per creare un server con java socket

## 2 Sistemi Distribuiti

### 2.1 Architetture

Esistono due livelli attraverso i quali studiare l'organizzazione di un sistema:

- **Architettura Software** (organizzazione logica): definisce quali sono le componenti del sistema, quali sono i paradigmi di comunicazione, i ruoli e le responsabilità di ogni componente.
- **Architettura di sistema** (organizzazione fisica);

### 2.1.1 Architetture software: layered architecture

Le componenti sono organizzate in livelli. Un componente al livello i può fare una chiamata ai livelli sottostanti j. Una componente i può fare una chiamata ai livelli superiori per **notificare l'occorrenza di un evento**. Un esempio

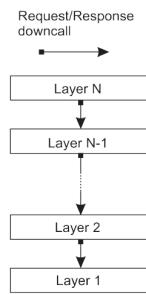


Figure 10: Layered architecture

la pila ISO/OSI. Un altro esempio quello generico dell'application layer che strutturato in 3 livelli:

- **Presentation layer**: gestisce le interazioni con gli utenti ;
- **Processing Layer**: svolge i calcoli e gestisce l'applicazione;
- **Data Layer**: gestisce il salvataggio dei dati.

### 2.1.2 Architetture software: Object-based architecture

Le **componenti sono oggetti** che interagiscono attraverso chiamate di metodo. Gli oggetti sono distribuiti su macchine differenti pertanto dobbiamo implementare delle chiamate remote. **Gli oggetti sono dotati di un'interfaccia e di uno stato** (facilmente sostituibili con oggetti con la stessa interfaccia). Vediamo l'architettura in dettaglio:

- L'oggetto sul server;

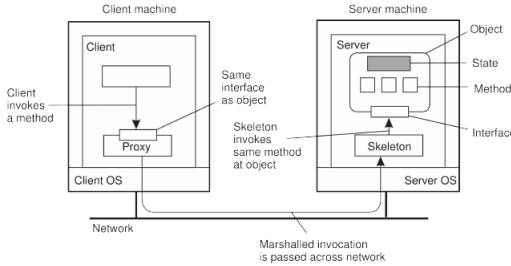


Figure 11: Layered architecture

- Il proxy implementa l’interfaccia dell’oggetto e resta sul client. Svolge il compito di eseguire il marshaling dei parametri necessari all’invocazione del metodo e di eseguire l’unmarshaling delle risposte contenenti i risultati;
- lo scheletro riceve la richiesta di invocazione, esegue l’unmarshaling dei parametri ed esegue la chiamata a metodo. (esempio java rmi).

### 2.1.3 Architetture software: Service oriented

Le componenti sono **servizi** che interagiscono attraverso i protocolli di rete. I servizi possono essere implementati da provider diversi utilizzando tecnologie diverse. Da questo punto di vista un’applicazione distribuita è una composizione di servizi che cooperano in armonia. Vediamo gli attori principali in questa architettura:

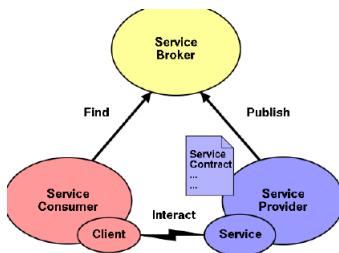


Figure 12: Layered architecture

- **Broker:** ha le informazioni su quali sono i servizi disponibili per gli utenti;

- **Provider:** fornisce il servizio e fornisce al broker le informazioni su quest'ultimo.
- **Consumer:** consulta il broker per sapere quali sono i servizi disponibili. Successivamente richiede al provider il servizio scelto.

Un altro concetto importante in questo tipo di architettura quello di **orchestration**. All'interno del sistema presente un **orchestratore** che gestisce l'interazione tra i servizi differenti

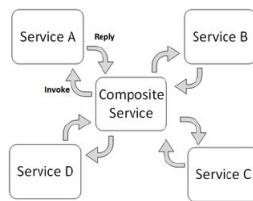


Figure 13: Layered architecture

#### 2.1.4 Restful architecture

In questo tipo di architettura un sistema distribuito una **collezione di risorse**. Vediamo le 4 caratteristiche principali

- Ogni risorsa viene identificata da un URI;
- Ogni servizio offre le stesse 4 operazioni
- esecuzione **stateless**
- Ogni messaggio completamente auto descrittivo ( contiene tutte le informazioni necessarie per processarne il contenuto).

Da notare che questa architettura :

- **Referentially coupled:** durante la comunicazione le componenti utilizzano riferimenti diretti al proprio partner di comunicazione.
- **Temporally coupoled:** per comunicare le componenti devono essere in esecuzione.

- A resource can be a singleton or a collection  
`https://api.example.com/resources/` collections  
`https://api.example.com/resources/item17` a single element
- A resource may contain sub-resources (hierarchical organization)
- There is no standard way for resource naming, but some best practices, e.g., not use query parameters to identify a resource but to provide parameters to an operation  
`https://api.example.com/resources/item17?format=json`

Figure 14: Risorse rest

Operation	Description
PUT	Create a new resource
GET	Retrieve the state of a resource
DELETE	Delete a resource
POST	Modify a resource

Figure 15: Operazioni rest

### 2.1.5 Publish subscribe architecture

Ha tre tipi di componenti:

- **Publisher:** invia messaggi in broadcast senza sapere se verrano ricevuti;
- **subscribers:** sta in ascolto per i messaggi che gli interessi e non ha nessuna conoscenza riguardo al publisher.
- **Broker:** intermediario tra i due.

Il sistema aperto, ci significa che non c' è un limite al numero di sub e pub, tuttavia il broker rappresenta il collo di bottiglia del protocollo.

Un **evento** consiste alla presenza di un nuovo dato disponibile. In questo caso il sistema può inoltrare la notifica insieme al dato oppure inoltrare solo la prima. Ricordiamo che il modello :

- **referentially decoupled:** le componenti non devono conoscersi per comunicare.
- **Temporally decoupled:** non devono essere entrambe attive per comunicare.

1. Topic-based: topics work as logical channels  
`myhome/groundfloor/livingroom/temperature`
2. Content-based: all data published is structured, thus, filters are constraints/predicates expressed on attributes  
`name=Acme* and value>20$`
3. Type-based: events are objects belonging to a specific type, which can encapsulate attributes as well as methods, thus, subscription filter on the type, e.g., any sub-type of Exception

Figure 16: Tipi di messaggi

### 2.1.6 Tuple space architecture

Le componenti comunicano tra di loro attraverso tuple che vengono salvate all'interno del tuple space (spazio condiviso).

- le componenti salvano tuple all'interno del tuple-space.
- per recuperare un tupla una componente deve fornire un pattern di ricerca che coincida con la tupla (`in(array,primes,int,int)`)
- il tuple space persistente: le tuple non vengono rimosse se non esplicitamente.
- produttore e consumatore non devono esistere nello stesso momento.

### 2.1.7 Architetture di sistema: client/server

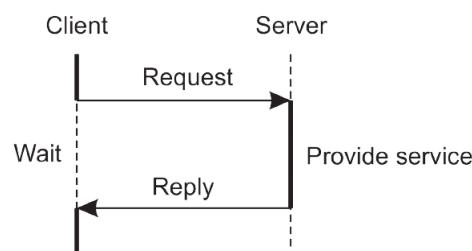


Figure 17: Architettura client server

Questa architettura utilizza NFS come modello di accesso remoto ai file:

- al client offerto accesso trasparente al file system gestito dal server

- Le operazioni offerte al client sono implementate sul server.
- ogni operazione su un file richiede la comunicazione con il server

In alternativa possiamo utilizzare un modello **upload/download**: il client scarica il file, esegue le operazioni ed esegue l'upload del file sul server quando ha terminato. Vediamo alcune organizzazioni tipiche:

- **Single tier**: l'applicazione sul server, il terminale permette solo di accedere al mainframe.
- **Two-tier**: il primo strato si trova sul client, il secondo sul server
- **Three-tier**: ogni layer su una macchina diversa.

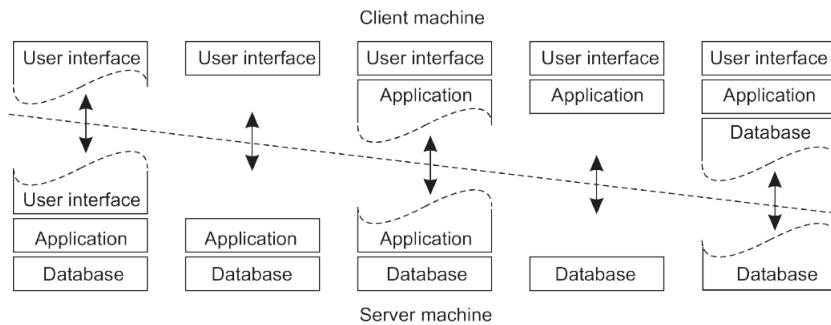


Figure 18: two tier

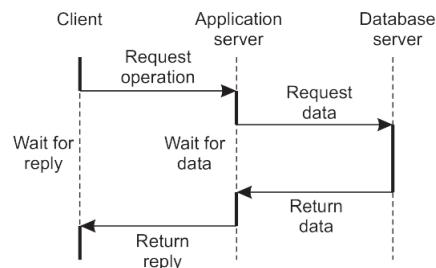


Figure 19: three tier

### 2.1.8 Architettura peer to peer

In questo tipo di architettura non abbia nessuna distinzione tra client e server: ogni nodo partecipa al sistema condividendo le proprie risorse. L'obiettivo quello di condividere risorse di un gran numero di partecipanti con lo scopo di svolgere un compito. I **peer** eseguono lo stesso programma e forniscono la stessa interfaccia. L'interazione dunque **simmetrica**

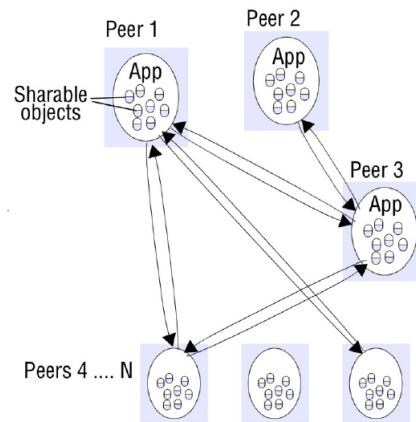


Figure 20: p2p architecture

### 2.1.9 P2P overlay network non strutturata

- ogni peer mantiene un lista locale di peer, quando un peer si unisce alla rete prima contatta uno di questi peer per farsi dare la lista dei partecipanti correnti al sistema.
- il peer scegli una partizione di peer a cui connettersi
- la topologia della rete risultante quella di un grafo casuale ovvero un grafo in cui un arco esiste solo con una certa probabilit.

Le **risorse** sono distribuite nei peer che partecipano al sistema: cercare una risorsa significa cercare i peers che sono responsabili per tale risorsa. Un peer conosce solo la sua lista dei peer locali e le risorse per cui essi sono responsabili. **Come pu un peer individuare le risorse che gli servono?** In un overlay network non strutturata si hanno due strategie possibili:

- **Floodind:** un peer passa la richiesta per una risorsa a tutti i suoi vicini. Un vicino pu fornire la risorsa se la possiede altrimenti inoltra la query a tutti i suoi vicini.
- **Random Walk** stesso procedimento ma casuale.

### 2.1.10 P2P overlay network strutturata

La rete ha una specifica topologia (anello, albero). Ogni risorsa del sistema associata ad una chiave, dunque il sistema salva coppie (**chiave, valore**) (DHT)

## 2.2 Jersey

Jersey è un framework per costruire **servizi web REST** (implementa le API **JAX-RS**). JAX-RS implementa API Java per i servizi REST ovvero una serie di annotazioni, classi e interfacce che possono essere utilizzate per esporre risorse POJO (assume http come protocollo di comunicazione sottostante). Concetti fondamentali:

```
@Path("mycalendar")
public class Calendar {

    @GET
    @Produces("text/plain")
    public String now(){
        Date nowDate = new Date();
        return nowDate.toString();
    }
}
```

Figure 21: p2p architecture

- **Resource class:** classe Java che utilizza annotazioni JS-RX
- **Root resource class:** una classe di risorse annotata con @path
- **Request method designator:** annotazione Java utilizzata per identificare un oggetto HTTP

- **Resource method:** un metodo di una risorsa annotato con un request method designator

```

Client client = ClientBuilder.newClient();
WebTarget target = client.target(BASE_URI)
    .path(RESOURCE)
    .queryParam("postId", "1");
Invocation.Builder request =
    target.request(MediaType.APPLICATION_JSON);
Response res = request.get();
System.out.println("Request status" + res.getStatus());
System.out.println(res.readEntity(String.class));

```

Figure 22: Esempio invocazione

**WebTarget** un costrutto che ci permette di definire una richiesta per una specifica risorsa, ne otteniamo così l'URI relativo e possiamo inoltre passare dei parametri per un'eventuale query avanzata. **InvocationBuilder** usa la richiesta formata attraverso WebTarget generando una richiesta http.

```
@Path("/users/{username}")
```

Figure 23: @path

```

public String getUser(@PathParam("username") String userName) {
    ...
}

```

Figure 24: @path

- **@path:** annotazione utilizzata per definire dove la classe ha il suo path di origine. **@pathparam** utilizzata nei parametri del metodo per indicare che quel parametro serve ad indicare una particolare risorsa nel path.
- **@GET, @PUT, @POST, @DELETE:** utilizzate per annotare i metodi

- **@Produces**: si utilizza per indicare che tipo di risorsa verrà prodotta per il client.
- **@Consumes**: associato ad un metodo @POST definisce il tipo dei parametri della richiesta.

`@GET, @PUT, @POST, @DELETE and @HEAD` decorate the Java method responsible to process the corresponding HTTP method

```
@DELETE
public void deleteRes() {
    ...
}
```

Figure 25: Annotazioni

```
@Path("/myResource")
@Produces("text/plain")
public class SomeResource {
    @GET
    public String doGetAsPlainText() {
        ...
    }

    @GET
    @Produces("text/html")
    public String doGetAsHtml() {
        ...
    }
}
```

Figure 26: Annotazioni

## 3 Communication Mechanisms

### 3.1 Middleware

Il **middleware** è un insieme di applicazioni e protocolli "general purpose" che risiedono all'interno del livello applicativo. Dunque un livello software che astrae dall'eterogeneità di rete, hardware, sistemi operativi e linguaggi di programmazione, con lo scopo di fornire interfacce comuni che assicurino

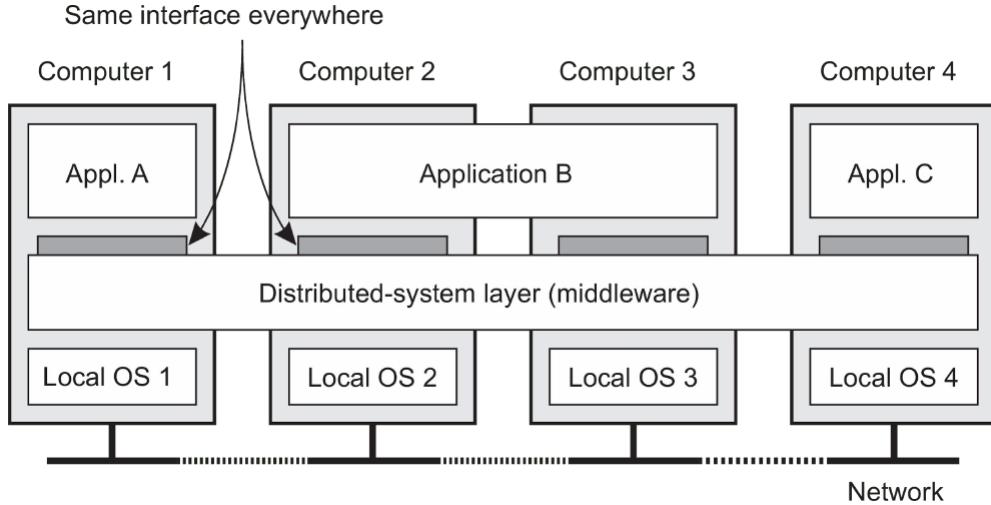


Figure 27: Livello Middleware

**modelli di comunicazione e di computazione uniformi.** Questo livello, dunque, costituisce un insieme di protocolli condivisi dalle applicazioni più 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'atomicità nelle transazioni. Stabiliscono se in un insieme di processi tutti hanno svolto una particolare operazione o se non è 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**.

### 3.2 Coordinazione diretta

Un tipi di comunicazione nella quale le componenti partecipanti sono:

- **Referentially coupled:** durante la comunicazione gli attori utilizzano riferimenti esplicativi 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).

### 3.3 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 modalità 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 risponde 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.

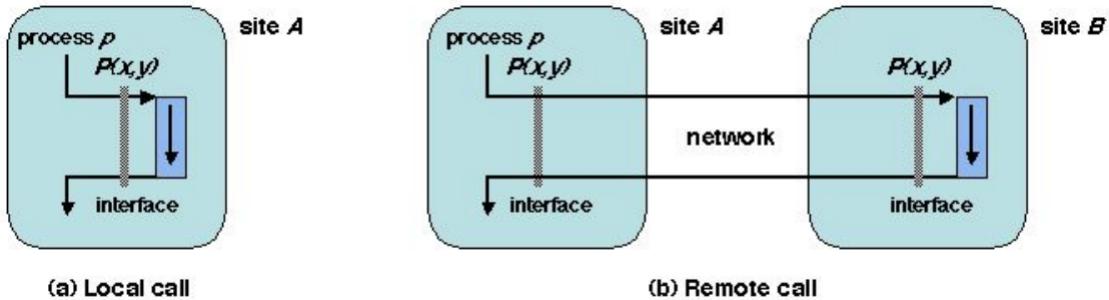


Figure 28: Chiamata a procedura locale vs remota

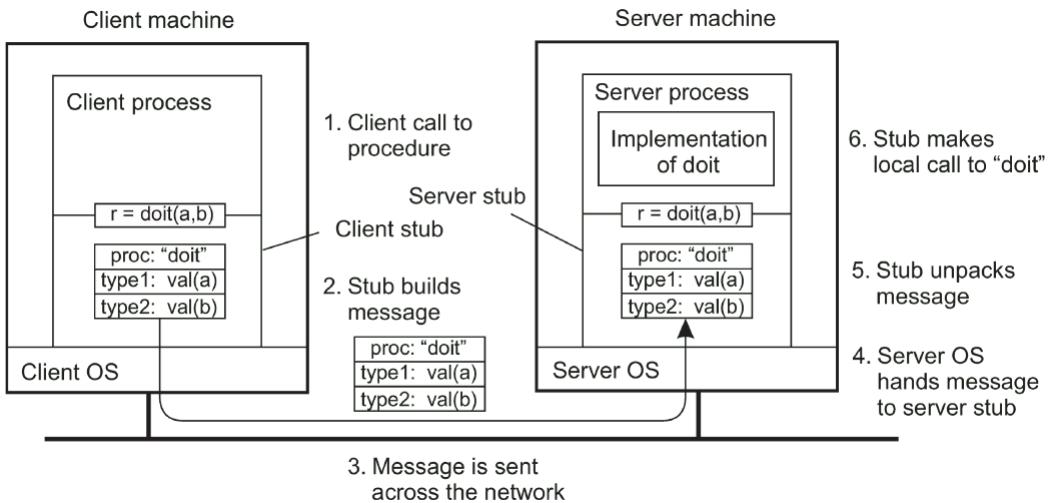


Figure 29: Funzionamento RPC

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

### 3.3.1 Passaggio di parametri

L'operazione di impacchettare parametri all'interno di un messaggio chiamata **marshaling**, il messaggio conterrà i parametri stessi e le informazioni necessarie al destinatario. Il principale problema è 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 può 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 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.

Il problema non si presenta qualora i riferimenti siano **globali**, ovvero quando hanno un significato sia per il server sia per il client.

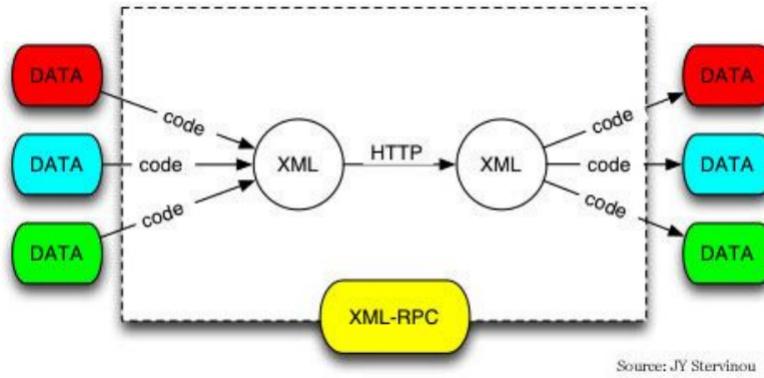


Figure 30: Xml

```
# 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 31: Marshaling in Java

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).

### 3.3.2 Implementare RPC

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

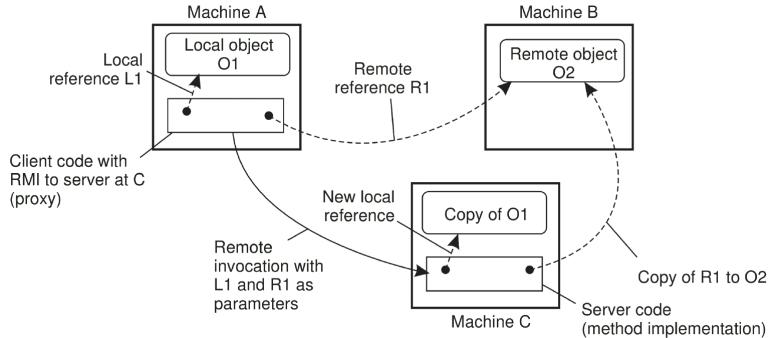


Figure 32: Oggetti remoti e locali

- **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**).

### 3.3.3 RPC Asincrono

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 sarà processata, nel frattempo il client può eseguire ulteriori operazioni evitando di sospendere la sua esecuzione. Il Server utilizza una funzione detta di **Callback** per consegnare il risultato al Client. L'asincronicità della comunicazione permette l'implementazione di un protocollo **Multicast**

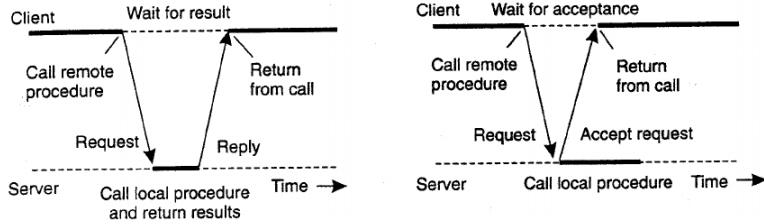


Figure 33: RPC tradizionale e asincrona

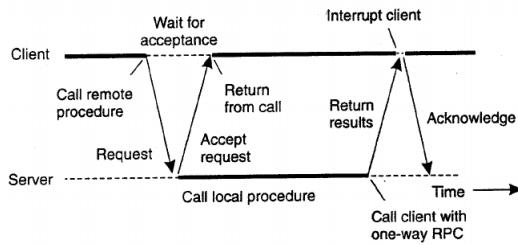


Figure 34: Callback

**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.

### 3.3.4 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.

## 3.4 Message Oriented Middleware

Questo modello di comunicazione prevede lo scambio di messaggi tra le entit participant. 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

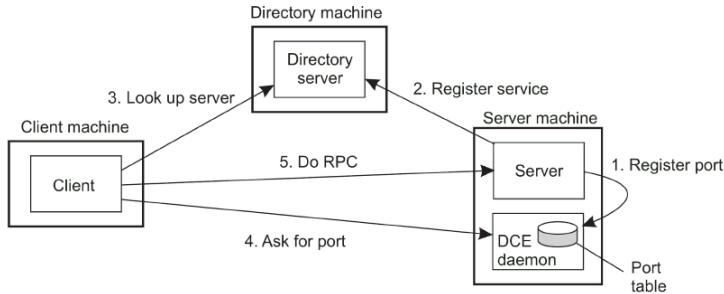


Figure 35: Binding

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

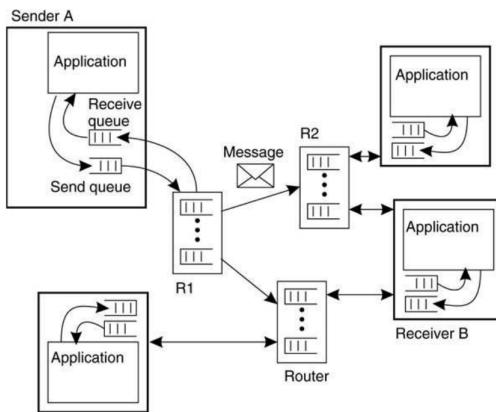


Figure 36: Code

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 è arrivato nella coda.

### 3.4.1 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 supporterà la comunicazione asincrona persistente.* In definitiva questi processi operano come

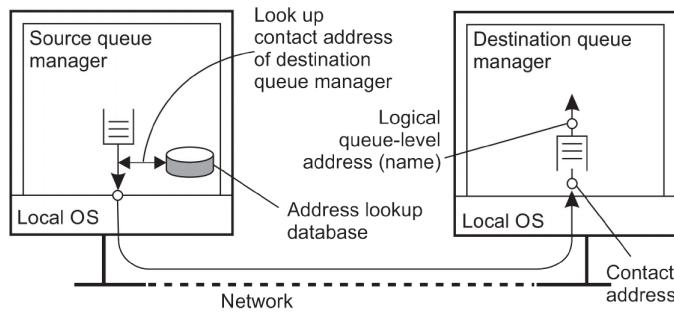


Figure 37: Queue Manager

**router** o **relay** inoltrando i messaggi ricevuti ad altri queue manager. In questo modo il sistema di queuing 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

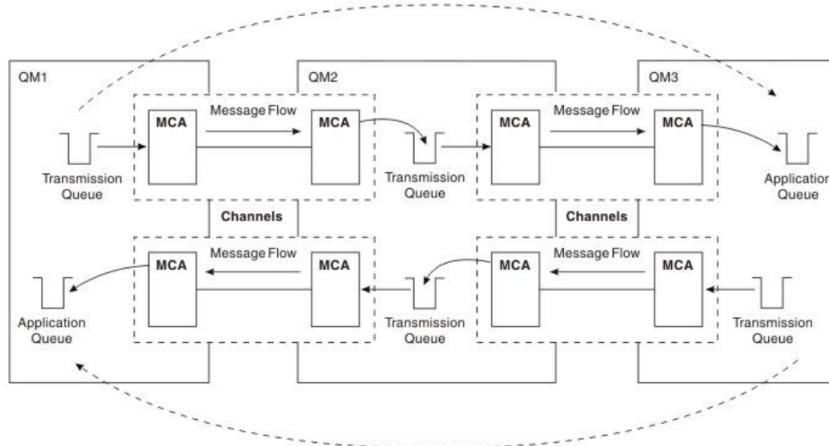


Figure 38: Overlay network

ogni entità 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 responsabilità di tenere traccia della topologia di rete e di aggiornare i binding (nome, indirizzo), mentre le altre entità partecipanti possiedono dei riferimenti statici al/ai router più vicino.

### 3.4.2 Eterogeneità: Message Brokers

I sistemi distribuiti possono essere eterogenei rispetto ai linguaggi utilizzati per realizzare le singole entità partecipanti. In questi casi è difficile definire un protocollo condiviso poiché assente alla base un accordo sul formato dei dati messaggi scambiati.

Un **Message Broker** si comporta come un gateway: 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:

## 3.5 Java RMI

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

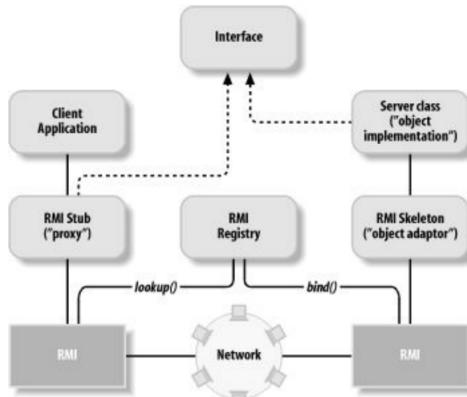


Figure 39: Architettura di RMI

modello presenta 4 entità principali:

- **Interfaccia**: utilizzata per definire la risorsa remota;
- **Server**: implementa la risorsa remota (che sarà 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 può 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 modalità 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 così 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 possibilità 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).

### 3.6 gRPC

gRPC è 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 è 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 più 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 è 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 necessarie 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 40: Definizione di un interfaccia con gRPC IDL

### Implementing the client

```
public class MyClient {

    ManagedChannel channel = ManagedChannelBuilder.forAddress("localhost",
8080).usePlaintext(true).build();

    HelloServiceGrpc.HelloServiceBlockingStub stub =
HelloServiceGrpc.newBlockingStub(channel);

    HelloResponse helloResponse = stub.hello(HelloRequest.newBuilder()
.setFirstName("Baeldung").setLastName("gRPC").build());

    System.out.println(helloResponse.getGreeting());
    channel.shutdown();
}
```

Figure 41: Client in grpc

Un canale una connessione virtuale per eseguire delle RPC, pu avere zero o piu connessioni con l’endpoint. Il canale deve essere associato ad un numero di porta e da un indirizzo.

## 4 Basic distributed algorithms

### 4.1 Contesto

Il **contesto** nel quale operiamo chiamato **ambiente distribuito**. Consiste in una collezione finita  $\epsilon$  di **entità** 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** (devo 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  $Stato \times Evento \rightarrow Azioni$  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.

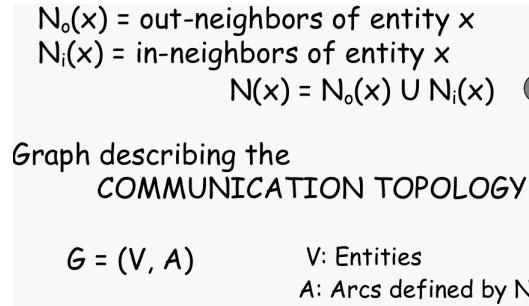


Figure 42: Come rappresentare la topologia di rete

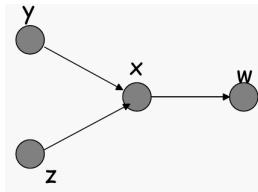


Figure 43: Topologia

#### 4.1.1 Assiomi

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



Figure 44: Labels

#### 4.1.2 Restrizioni

Si possono definire ulteriori proprietà o capacità in relazione ai compiti e agli obiettivi che il sistema distribuito si pone 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 assenza 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 restriction:**
  - **Guaranteed delivery:** ogni messaggio inviato viene recapitato al mittente non corrotto;
  - **Partial reliability:** garantisce l'assenza di fallimenti in futuro;
  - **Total reliability:** non ci sono stati fallimenti e non ce ne saranno.
- **Strongly connected:** il grafo  $g$  che rappresenta la topologia forte-mente connesso.
- **Knowledge restriction**
  - conoscenza del numero di nodi;
  - conoscenza del numero di link;
  - conoscenza del diametro.

#### 4.1.3 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 occorra. 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.

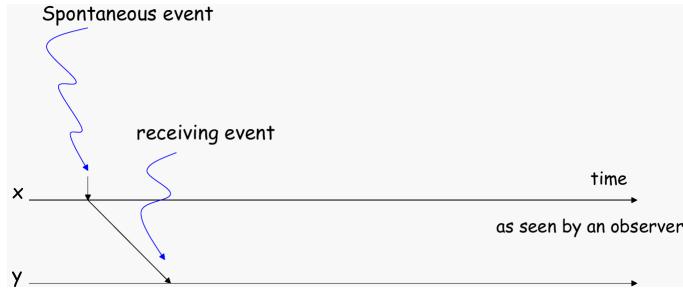


Figure 45: Stato x Evento

#### 4.1.4 Livelli di Conoscenza

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

I **tipi di conoscenza** includono quelle **topologiche, metriche** (numero di nodi, diametro, eccentricità), **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.

## 4.2 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**.

### 4.2.1 Flooding

**Assunzioni:** (BL, CN, TR) Bidirectional link, Connectivity (ogni entità 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. Ovviamente il mittente viene escluso dalla lista dei nodi ai quali inoltrare l'informazione ricevuta. L'algoritmo gode della proprietà di **Termination**:

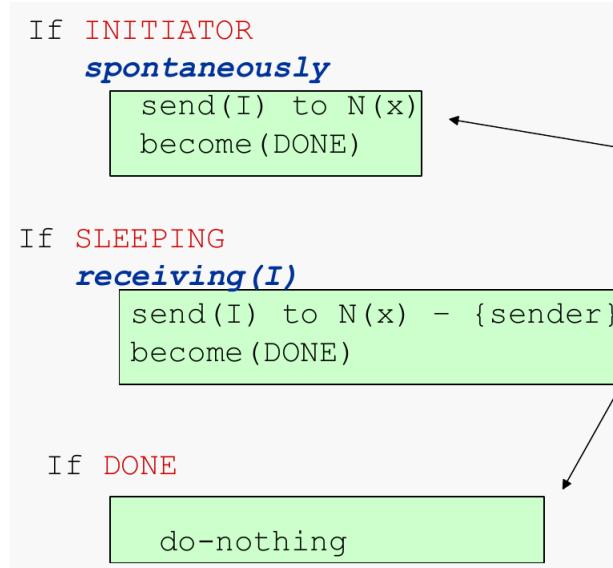


Figure 46: Algoritmo Flooding

**nation:** l'algoritmo termina in tempo finito (local termination quando lo stato *done*). Garantita dal fatto che il grafo connesso e che vale la proprietà di total reliability. Il caso peggiore si presenta quando il **grafo completo**. Per quanto riguarda la **complessità dei messaggi**: vengono scambiati 2

messaggi per ogni link

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

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 complessità ideale in tempo abbiamo:

$$r(x) = \text{Max}_x(d(x, y)y \in \epsilon) = \text{eccentricity} \leq \text{Diameter}(G) \leq n - 1$$

La complessità in tempo dipende dunque dalla massima distanza dell'iniziatore con un altro nodo della rete. Il caso peggiore risulta quindi essere quello nel quale questa distanza è il diametro. con:

$$\text{Diameter}(G) = \text{Max}_x(r(x))$$

### 4.3 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:

#### 4.3.1 Broadcast in un Hypercube

Per  $k = 1$  un hypercube è 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 dimensione  $H_{k-1}$  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 diviene  $k$ . Le etichette sono simmetriche rispetto ai nodi connessi dal link

*Ricordiamo che i nomi dei nodi sono utilizzati solo a scopo descrittivo e non sono conosciuti dalle entità. Al contrario i nomi delle etichette sono noti alle entità per l'assioma di local orientation.*

Vediamo la **complessità** del flooding per questa particolare topologia. Un hypercube di dimensione  $k$  ha  $n = 2^k$  nodi. Ogni nodo ha  $k$  link, pertanto il **numero totale di link** :

$$m = nk/2 = O(n \log(n))$$

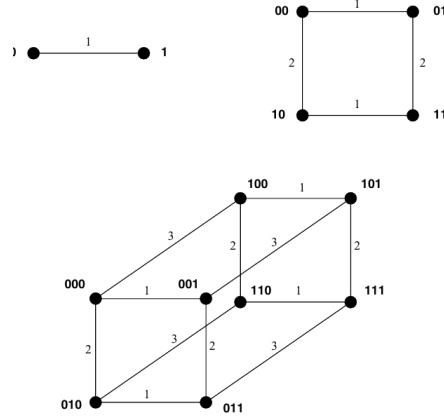


Figure 47: Hypercube

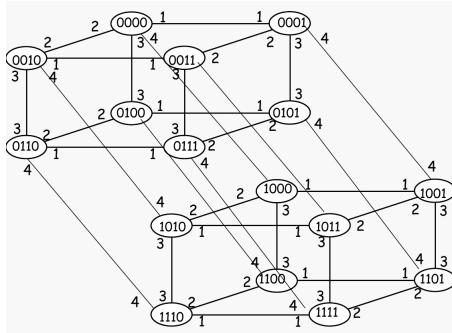


Figure 48: Costruzione hypercube

(si divide per due perch ogni link condiviso da due entit) il costo del flooding pertanto:

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

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 (vedi immagine).

**Dimostrazione:**

- Consideriamo due nodi  $x$  e  $y$  con etichette

$$H_x[x_k, x_{k-1}, \dots, x_0]$$

$$H_y[y_k, y_{k-1}, \dots, y_0]$$

. Se  $x \neq y$  allora abbiamo un numero di cifre  $t > 0$  per cui le due etichette differiscono.

- Consideriamo le etichette leggendole da sinistra a destra

$$j_1, j_2, \dots, j_t$$

con

$$j_i > j_{i+1}$$

- Se consideriamo adesso la sequenza di nodi

$$v_0, v_1, \dots, v_t$$

si ha che  $v_i$  differisce da  $v_{i+1}$  solo di una cifra ovvero la posizione  $j_{i+1}$ .

- Se  $v_0 = x$  e  $v_t = y$  si ha che le loro etichette differiscono in  $t$  posizioni e che il cammino di etichette dunque decrescente

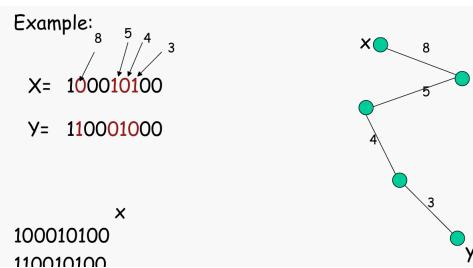


Figure 49: Lemma

Il risultato che il messaggio crea uno **spanning tree** e ogni nodo raggiunto da tale messaggio. La complessità risulta essere quella ottimale ( $n-1$ ) perché le entità ricevono l'informazione solo una volta. La complessità ideale in tempo  $k$  poiché l'eccentricità di ogni nodo  $k$ .

### 4.3.2 Broadcast in un grafo completo

Il flooding ha complessità generica:

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

Per quanto riguarda il flooding in un grafo completo risulta essere:

$$(n - 1)$$

Infatti poiché tutti sono connessi con tutti, l'iniziatore deve inviare l'informazione solo una volta.

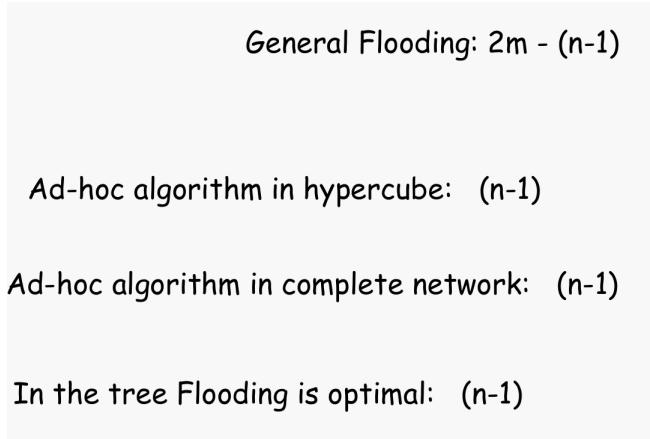


Figure 50: Complessità flooding in reti particolari

### 4.3.3 Lower bound

Torniamo al caso generale del broadcast e cerchiamo una limitazione inferiore per la sua complessità.

**Teorema:**

*Ogni algoritmo di broadcast richiede (assunto iniziatore unico), nel caso peggiore,  $O(m)$  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. Prendiamo quindi un grafo  $G$  nel quale esiste un arco, definito dai nodi  $x$  e  $y$ , sul quale non si scambiano messaggi.

Creiamo da  $G$  un grafo  $G'$  eliminando l'arco tra  $x$  e  $y$  aggiungendo un nodo  $z$  e collegando il nodo  $z$  con due archi a  $x$  e  $y$ . Se eseguiamo l'algoritmo con gli stessi delay di consegna e stessi stati,  $x$  e  $y$  non comunicano mai con  $z$  che non riceve pertanto l'informazione. ASSURDO.

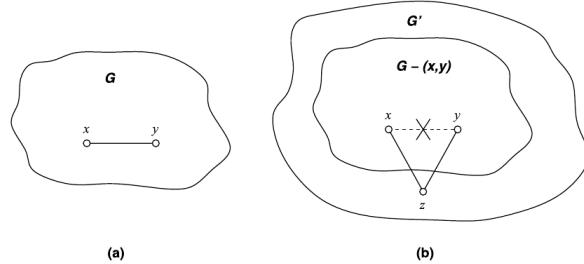


Figure 51: Dimostrazione lower bound broadcast

## 4.4 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**.

### 4.4.1 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 communication 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'entità  $x \neq s$  risponde "yes" solo la prima volta ed in questa occasione pone a tutti i suoi vicini la stessa domanda, altrimenti risponde "no".
3. Ogni entità termina quando ha ricevuto una risposta da tutti i vicini.

PROTOCOL Shout

- Status:  $\mathcal{S} = \{\text{INITIATOR}, \text{IDLE}, \text{ACTIVE}, \text{DONE}\}$ ;  
 $\mathcal{S}_{\text{INIT}} = \{\text{INITIATOR}, \text{IDLE}\}$ ;  
 $\mathcal{S}_{\text{TERM}} = \{\text{DONE}\}$ .
- 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 52: Algoritmo protocollo shout

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

#### 4.4.2 Correttezza

Sappiamo che flooding corretto e dunque sappiamo che ogni entità riceverà Q e che per costruzione risponderà *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, è uno spanning tree di G.**

Sappiamo che:

- Se  $x$  è un tree-neighbor di  $y$  allora  $y$  è un tree-neighbor di  $x$ ;
- Se  $x$  invia *yes* ad  $y$ , allora  $x$  è un tree-neighbor di  $y$  ed è 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).

Poiché ogni entità  $x \neq s$  invia un solo *yes* allora  $G^1$  contiene tutte le entità di G, è connesso e non contiene cicli e dunque è uno spanning tree di G.

Importante ricordare che Shout termina per **terminazione locale** ovvero che ogni entità 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).

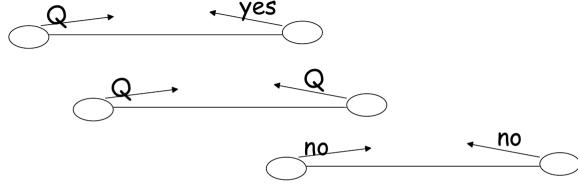
#### 4.4.3 Costo computazionale

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

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

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

Possible situations



Impossible situations



Figure 53: Shout: possibili situazioni

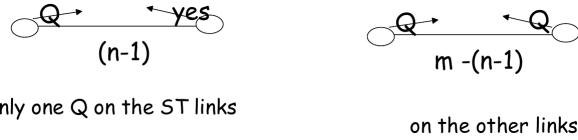


Figure 54: Totale dei messaggi Q (sei mio vicino?)

$$\begin{aligned} \text{Total: } & 2(m - (n-1)) + (n-1) \\ & = 2m - n + 1 \end{aligned}$$

Figure 55: Totale dei messaggi Q: formula

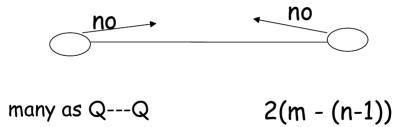


Figure 56: Totale dei messaggi no



Figure 57: Totale dei messaggi yes

#### 4.4.4 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

$$\begin{aligned}
 & 2m - n + 1 + 2(m - (n-1)) + n-1 \\
 & = 2m - n + 1 + 2m - 2n + 2 + n - 1 \\
 & = 4m - 2n + 2
 \end{aligned}$$

Figure 58: 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'entità risponde *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 complessità si riduce a:

$$Messages(SHOUT) = 2m$$

Un'altra possibile miglioria implementata con lo scopo di ottenere **terminazione globale**

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

```

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

Figure 59: 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.

<b>INITIATOR</b> <i>Spontaneously</i> <pre> root:= true Tree-neighbours := { } send(Q) to N(x) counter:= 0 <b>ack-counter:= 0</b> become ACTIVE </pre>	<b>IDLE</b> <i>receiving(Q)</i> <pre> root:= false parent := sender Tree-neighbours := {sender} send(yes) to sender counter := 1 ack-counter:= 0 if counter =  N(x)  then     <b>CHECK</b> else     send(Q) to N(x) - {sender} become ACTIVE </pre>
---	--

Figure 60: Algoritmo Shout con global termination 1

### ACTIVE (cont)

```

receiving(Ack)
    ack-counter:= ack-counter +1
    if counter = |N(x)| // indicate tree-neighbors is done
        if root then
            if ack-counter = |Tree-neighbours|
                send(Terminate) to Tree-neighbours
                become DONE
            else if ack-counter = |Tree-neighbours| - 1
                send(Ack) to parent

receiving(Terminate)
    send(Terminate) to Children
    become DONE

// Children is Tree-neighbours - {parent}

```

Figure 61: Algoritmo Shout con global termination 2

### CHECK

```

If I am a leaf
(* that is: Children:= Tree-neighbours - {parent}
   if Children = emptyset *)
    send(Ack) to parent

```

Figure 62: Algoritmo Shout con global termination

#### 4.4.5 Iniziatore multiplo

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连通的. Questo risultato parti-

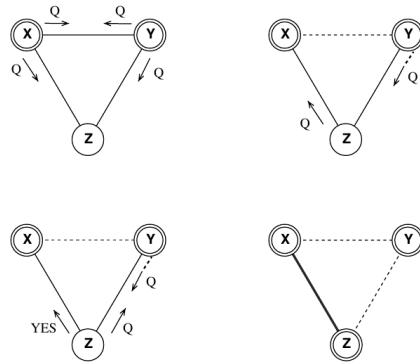


Figure 63: Iniziatori multipli in Shout

colare avvallato dal **risultato di impossibilità**:

**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 entità hanno lo stesso codice e perciò, iniziando simultaneamente nello stesso stato, riceveranno gli stessi messaggi e svolgeranno le stesse computazioni, trovandosi sempre negli stessi stati (tutte le entità 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 entità 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**.

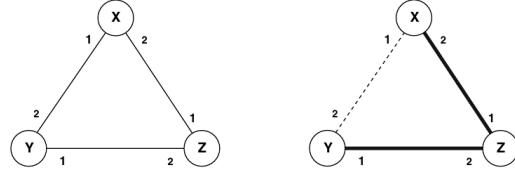


Figure 64: Prova risultato impossibilit

#### 4.4.6 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 che viene inoltrato finch non si incontra un nodo gi visitato. a quel punto il token restituito al mittente che prover a sua volta a inoltrarlo ad uno dei suoi vicini non visitati se ne ha. L'algoritmo termina quando il token si trova in un nodo che ha solo backedge.

- Quando un nodo viene visitato per la prima volta, tiene traccia di chi il mittente ed, crea una lista dei vicini non visitati, ed inoltra il token ad uno di questi.
- Quando un vicino riceve il token, se gi stato visitato marca l'arco come **back-edge** (eliminando il mittente dalla lsita dei nodi non visitati)e restituisce il token al mittente, altrimenti inoltra sequenzialmente il token a tutti i suoi vicini .
- 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.

In altre parole l'albero dei vicini di un'entit x composto dalle entit dalle quali ha ricevuto un messaggio di return

FARE IMMAGINE CON SLIDE DA 30 a 41

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, dopo aver provato ad inoltrare il messaggio ai suoi vicini, o back se gi visitato). Dunque:

$$2m = O(m)$$

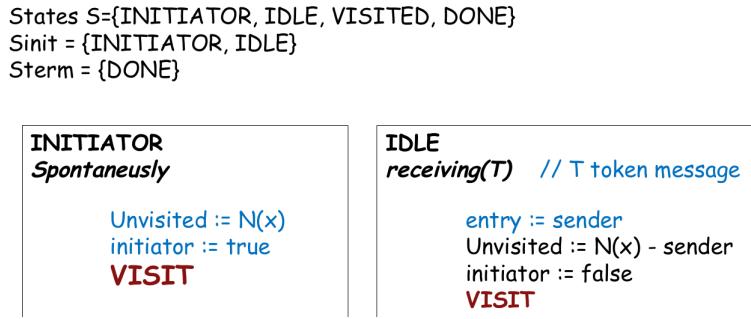


Figure 65: Algoritmo DFS 1

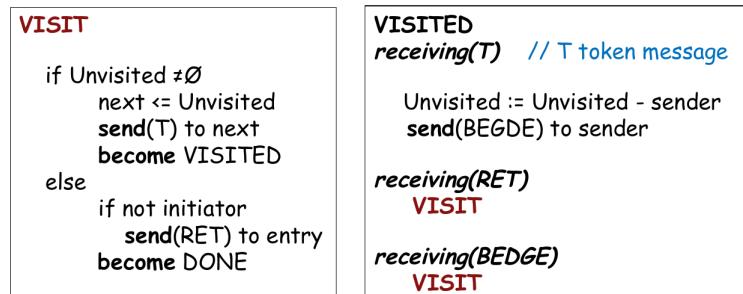


Figure 66: Algoritmo DFS 2

con

$$O(m) = \text{lowerbound}$$

La complessità in messaggi del protocollo DF ottimale ( $2m$ ) infatti dello stesso ordine del suo lower bound. La complessità in tempo la stessa di quella in messaggi poiché il protocollo **sequenziale**.

$$2m = O(m)$$

con

$$O(n - 1) = \text{lowerbound}$$

Tuttavia considerando il tempo la complessità  $2m$  può essere di magnitudine molto più grande rispetto al suo lower bound ( $n-1$ ). Ad esempio in un grafo completo

$$2m = n^2 - n$$

#### 4.4.7 DF+

L’obiettivo quello di migliorare la complessità in tempo , passando da un protocollo completamente sequenziale (in una esecuzione sincrona ad ogni istante di tempo si invia un solo messaggio ed ogni messaggio richiede un’unità di tempo ). 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 complessità in messaggi abbiamo che ogni entità riceve un **token** ed invia un **return**

$$2(n - 1)$$

Inoltre ogni entità invia 1 **visited** a tutti i vicini tranne al mittente (lo stesso vale per i messaggi di ACK quindi la sommatoria sotto da moltiplicarsi per 2).

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

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

$$4n - 2$$

#### 4.4.8 DF++

Possiamo fare a meno dei messaggi di ACK ?

In questo caso una volta aver inviato messaggi **visited** una unit x invia direttamente il token ad uno dei suoi vicini senza attendere i messaggi di ACK. Supponiamo che il token arrivi ad un vicino di x per la prima volta, inviato da un'altra entità y. Se il delay sui messaggi visited di x è considerevole può capitare lo scenario nel quale z non sa che x gli è stato visitato e gli invia il token. Entrambi possono capire l'errore e far finta che non sia avvenuto:

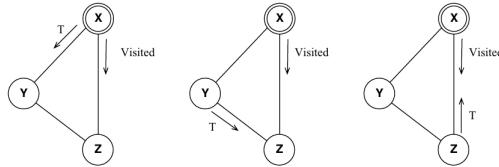


Figure 67: Algoritmo DF++

z continua con il forwarding del token togliendo x dalla lista dei nodi non visitati, mentre x scarta il token.

Vediamo la **complessità**: i messaggi **token** e **return** sono sempre

$$2(n - 1)$$

e dunque anche i messaggi di **visited** sono sempre

$$2m - (n - 1)$$

Il numero degli errori può essere molto grande, infatti potrebbero esserci 2 errori su ogni back edge. I back edge sono  $m - (n - 1)$  e i messaggi sbagliati che si muovono su questi archi sono 2 (un token e un visited):

$$2m - 2n + 2$$

. Il totale :

$$2n - 2 + 2m - n + 1 + 2m - 2n + 2 =$$

$$4m - n + 1$$

Per quanto riguarda la complessità in tempo abbiamo che i messaggi Token e visited sono inviati nello stesso momento (non aspettiamo più gli ack) così come

i return (qualora non ci sia nessun vicino a cui inoltrare il token). Pertanto risparmiamo  $2n$  unit temporali da sottrarre alla complessità in tempo di DF+:

$$4n - 2 - 2n = 2n - 2$$

Dovremmo considerare i possibili errori ma la complessità ideale in tempo calcolata in un contesto di sincronia pertanto gli errori dovuti ai delay non si contano.

#### 4.4.9 Smart traversal

Osservazione: ogni algoritmo che risolve il problema dell'attraversamento di un albero risolve anche il problema del broadcast (anche se magari non molto efficiente, mentre non vero il contrario). Per esempio flooding viola il requisito di *sequenzialità* che si ha per una visita di un albero. Possiamo utilizzare duqnue algoritmo shout che di fatto un flood + reply (dunque un broadcast) prima di eseguire la visita in profondit:

- si costruisce uno spanning tree con **shout+**  $2m$
- Si esegue visita in profondit.  $2(n-1)$
- per un totale di  $2(m + n - 1)$

	Messages	Ideal Time
DF:	$2m$	$2m$
DF+:	$4m$	$4n - 2$
DF++	$4m-n+1$	$2n-1$
Smart	$2m+2n-2$	$2n+d-1$

Figure 68: Complessità DF a confronto

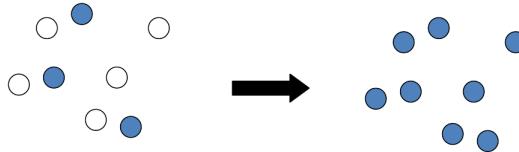


Figure 69: Complessit DF a confronto

#### 4.4.10 Wake-up

I sistemi distribuiti svolgono operazioni che potenzialmente richiedono la partecipazione attiva di tutti i nodi della rete. Abbiamo dunque bisogno di un procedura preliminare che al suo completamento restituisce una topologia nella quale tutti i nodi sono **awake**: si utilizza il protocollo **Wake-up**. Il protocollo inizializzato dalle unit che sono indipendentemente già attive e dunque si prevede la possibilità di **iniziatori multipli**.

- **Broadcast** un wake up con multipli iniziatori
- wake up un broadcast con la possibilità di avere iniziatori multipli

Per quanto riguarda la complessità si ha dunque:

- **lower bound**:  $O(m)$  (in tempo  $O(d)$ );
- si utilizza un'algoritmo di flooding generico  $2m - n + 1$ , considerando  $k$  iniziatori

$$2m - n + k^*$$

Per quanto riguarda **topologie specifiche**:

- **Alberi**: nel caso di un iniziatore avremmo un costo in messaggi di  $(n-1)$ , se ognuna delle entità inizia in modo indipendente abbiamo il caso pessimo  $2(n-1) - n + n = 2(n-1)$ . In generale la complessità

$$2m - n + k^* = 2(n-1) - n + k^* = n + k^* - 2$$

- **Hypercube**: sappiamo che possiamo sfruttare le proprietà degli hypercubes per realizzare un algoritmo di broadcast che ha complessità  $O(n)$ . Tuttavia per quanto riguarda il broadcast (iniziatori multipli) la complessità è di  $O(n \log n)$ .

COMPLETE GRAPH											
Broadcast				Wakeup							
Flood	Specific		$O(n^2)$	Flood	Specific		$\Omega(n^2)$				
	$O(n)$				Need additional assumptions to reduce the complexity						
HYPERCUBE											
Broadcast				Wakeup							
Flood	Specific		$O(n \log n)$	Flood	Specific		$\Omega(n \log n)$				
	$O(n)$										

Figure 70: Complessità wake up in reti speciali

## 4.5 Computazione negli alberi

In questa sezione analizziamo 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 è una **foglia** o un **nodo interno** (se ha un solo vicino o più di uno).

### 4.5.1 Saturation

La **Saturazione** è una tecnica base che può 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 dipende 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 entità (con lo scopo di ottenere *local termination*). Un protocollo "troncato" come questo chiamato **plug-in** (un protocollo nel quale non tutte le entità entrano in stato terminale). Per far sì che diventi un protocollo necessario definire ulteriori azioni da svolgere. **Il protocollo può essere avviato da un qualsiasi numero di initiatori**. Vediamo le fasi in dettaglio:

1. **Attivazione**: questa fase è 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 già 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 communication delay**)

2. **Saturazione:** questa fase inizializzata dalle foglie che inviano il messaggio **M** di saturazione al loro unico vicino (il parent), entrando così 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
        /* special case if
        I am a leaf */
        Prepare_Message; // M := "Saturation"
        parent << Neighbours;
        send(M) to parent;
        become PROCESSING;
    else
        become ACTIVE;


```

Figure 71: Algoritmo Saturazione 1

#### 4.5.2 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*

```

AVAILABLE
Receiving(Activate)
    send(Activate) to N(x) - {sender};
    Initialize;
    Neighbours:= N(x);
    if |Neighbours|= 1 then
        Prepare_Message; //M := "Saturation"
        parent << Neighbours;
        send(M) to parent;
        become PROCESSING;
    else
        become ACTIVE;

```

Figure 72: Algoritmo Saturazione 2

<b>ACTIVE</b>	I haven't started the saturation phase yet
<b>Receiving(M)</b>	Process_Message; Neighbours:= Neighbours - {sender}; if  Neighbours  = 1 then Prepare_Message; //M := "Saturation" parent << Neighbours; send(M) to parent; become PROCESSING;
<b>PROCESSING</b>	I have already started the saturation phase
<b>receiving(M)</b>	Process_Message; become SATURATED; // Resolve

Figure 73: Algoritmo Saturazione 3

**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  $d(x, y) \geq 2$ . 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:** impredibile quale coppia di nodi divenga saturata, questo per via dei delay di consegna dei messaggi ( ovviamente incide anche la scelta degli iniziatori).

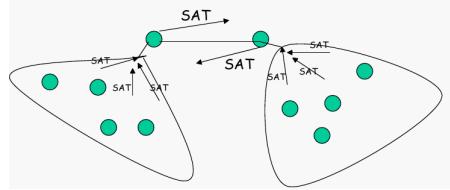


Figure 74: Algoritmo Saturazione: prova

#### 4.5.3 Complessit

- La fase di attivazione un wake-up per cui abbiamo che

$$n + k^* - 2$$

- Durante la fase di saturazione abbiamo che per ogni arco si trasmette un solo messaggio saturation  $n - 1$  tranne sull'arco che collega i due saturati

$$n - 1 + 1 = n$$

- per un totale di

$$2n + k^* - 2$$

Nel caso pessimo si ha che gli iniziatori sono n:

$$2n + n - 2$$

#### 4.5.4 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 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.

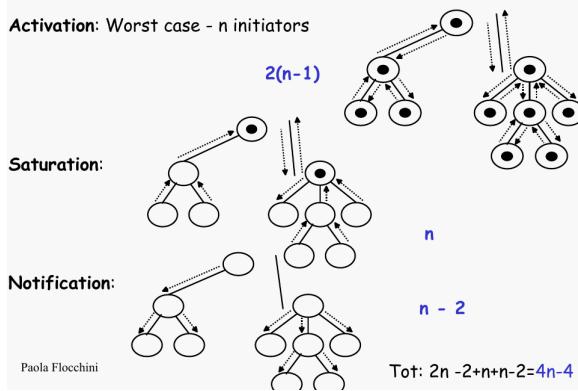


Figure 75: Algoritmo Saturazione: complessità caso peggiore

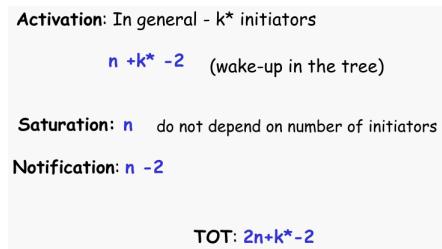


Figure 76: Algoritmo Saturazione: complessità caso generale con  $n$  iniziatori

Il problema può 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'esistenza 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 che diviene un fase di **notification** :

#### 4.5.5 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.

$$\begin{array}{c}
 \text{Saturation + Minimum Notification} \\
 \downarrow \qquad \qquad \qquad \downarrow \\
 2n+k^*-2 \qquad n - 2 \\
 \hline
 3n + k^* - 4
 \end{array}$$

Figure 77: Ricerca minimo con Saturazione: complessità

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 78: 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 79: Algoritmo computazione distribuita di funzioni 1

## 4.6 Leader Election

In molte applicazioni distribuite si richiede che una singola entità si comporti temporaneamente come un controller centrale che coordina l'esecuzione dei task. Il problema dunque quello di modificare la configurazione iniziale del

```

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 80: 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 81: Computazione distribuita di funzioni: complessit

*sistema, nel quale tutte le entit sono nello stesso stato, in una configurazione finale nella quale tutte le entit sono nello stesso stato meno che una: il Leader.*

#### 4.6.1 Risultato di impossibilit

Ricordiamo le restrizioni: **Bidirectional Link**, **Connettivit**, **Total Reliability**.

*Non esiste un protocollo deterministico per il problema della leader election che termini sempre in tempo finito.*

- Due entit x e y sono inizialmente nello stesso stato (available).
- Se un protocollo che risolve tale problema deve farlo in tutte le possibili configurazioni di **delay di consegna** dei messaggi. Consideriamo una comunicazione **sincrona** (delay di consegna unitari): se entrambe le entit iniziano l'esecuzione del protocollo nello stesso momento allora

essi scambieranno gli stessi messaggi ed eseguiranno dunque le stesse regole, terminando entrambe nello stesso stato.

- Se uno dei due diventa leader anche l'altro lo diventa contraddicendo il requisito per il quale al termine del protocollo dobbiamo avere un solo iniziatore.
- P non una soluzione valida.

Dobbiamo introdurre dunque altre restrizioni che permettano di rompere la simmetria che si verifica nel comportamento delle entit. Si potrebbe pensare di introdurre la restrizione di **UI+** ma in questo modo sposteremmo il problema (sarebbe come dire che riusciamo sempre ad eleggere un leader se c' n' gi uno). L'idea dunque quella di poter distinguere in modo univoco le entit dotandole di un **identificativo unico** (o nome globale).

In questo modo si ha quello che chiamiamo **standard set for election**.

#### 4.6.2 Election negli alberi

Vediamo alcune delle possibili strategie:

- Trovare il minimo (con l'introduzione degli ID la ricerca del minimo di fatto un algoritmo di leader election).
- Trovare il minimo tra gli iniziatori
- Costruire uno spanning tree con radice.

Si pu vedere che eleggere il minimo iniziatore all'interno dell'albero ha la stessa complessit di trovare il valore minimo utilizzando la tecnica della saturazione. La complessit in messaggi la seguente:

$$3n + k^* - 4 \leq 4n - 4$$

Per quanto riguarda la strategia che vede la **costruzione di un albero con radice** possiamo anche in questo caso applicare la tecnica della saturazione (ricordiamo che abbiamo gi una topologia ad albero dobbiamo solo eleggere la radice). Ricordiamo che utilizzando la *full saturation* si ottengono inizialmente due soli nodi saturati. Il risultato che avremo un albero con due radice: dobbiamo scegliere una delle due. **In poche parole diciamo che il problema di eleggere un leader tra tutti i nodi viene ridotto al**

problema di eleggere il leader tra due nodi (non ci importa che questi abbiano effettivamente il valore minimo).

La complessità in messaggi risulta analoga a quella relativa al protocollo **full**

```

SATURATED
    Receiving (Election, id*)
    begin
        if id(x) < id* then
            become LEADER;
        else
            become FOLLOWER;
        endif
        send ("Termination") to N(x) - {parent};
    end

PROCESSING
    Receiving ("Termination")
    begin
        become FOLLOWER;
        send ("Termination") to N(x) - {parent};
    end

Procedure Resolve
begin
    send ("Election", id(x)) to parent;
    become SATURATED;
end

```

Figure 82: Root Election

**saturation con notification.**

$$M[Tree : ElectRoot] = 3n + k^* - 2 \leq 4n - 2$$

Volendo fare un'analisi più (con granularità più) fine dobbiamo considerare i **bit spesi** per ogni messaggio. Da questo punto di vista l'algoritmo per l'elezione della radice è molto migliore rispetto a quello di elezione del minimo: solo gli n messaggi della fase di saturazione trasmettono un valore, gli altri sono semplici segnali.

#### Bit Complexity

$$\begin{aligned} \text{Elect\_min: } & n(c + \log id) + c(2n + k^* - 4) \\ \text{Elect\_root: } & 2(c + \log id) + c(3n + k^* - 2) \end{aligned}$$

$$c = \text{bit of header} \quad \log id = \text{bit of payload}$$

Figure 83: Bit complexity

#### 4.6.3 Leader election in un anello: All the Way

Consideriamo ora un'altra topologia classica nel contesto dei sistemi distribuiti: l'**anello**. Le proprietà caratteristiche di un anello sono le seguenti:

- Numero delle entit = numero link;
- Topologia simmetrica (tutti i nodi sono identici)
- Ogni entit ha due vicini

Il primo algoritmo trattato **all the way**. L'idea la seguente:

- Quando un'entit inizia la sua esecuzione sceglie uno dei due vicini ed invia un messaggio Election contenente il suo ID.
- Quando un'entit riceve un messaggio da uno dei due vicini, lo inoltra all'altro vicino. Inoltre invia un altro messaggio (se non l'ha già fatto) con il proprio ID.
- Ogni entit tiene traccia del più piccolo valore ricevuto.

In poche parole ogni entit invia un messaggio che attraversa tutta la topologia. ne consegue, ovviamente, che prima o poi tutte le entit vedano l'ID di tutte le altre entit (finite communication delay, total reliability).

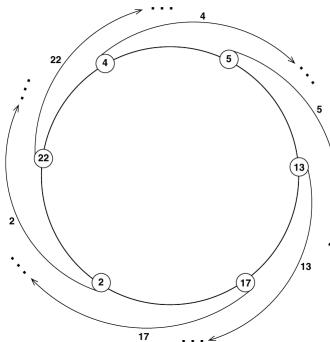


Figure 84: All the way

#### 4.6.4 All the way: correttezza e terminazione

Idealmente potremmo dire che ogni entità termina quando riceve indietro il proprio id. Questo ragionamento suggerito dal fatto che, considerando un nodo  $x$ , ogni messaggio inviato da un suo vicino qualsiasi deve attraversare un numero minore di nodi per raggiungerlo. Questo ragionamento risulta incorretto a meno di introdurre un'ulteriore assunzione: **message ordering**.

```

States: S={ASLEEP, AWAKE, FOLLOWER, LEADER}
S INIT={ASLEEP};
S TERM={FOLLOWER, LEADER}.
IR;Ring          ASLEEP

Spontaneously
INITIALIZE
become AWAKE

Receiving(`Election', value, counter)
INITIALIZE
send(`Election', value, counter+1)
to other
count:= 0
size:= 1
known:= false
send("Election",id(x),size) to right;
min:= id(x)

```

Figure 85: All the way 1

```

AWAKE
Receiving ("Election", value, counter)
If value <> id(x) then
    send ("Election",value,counter+1) to other
    min:= MIN{min,value}
    count:= count+1
    if known = true then
        CHECK
    endif
else
    ringsize:= counter
    known:= true
    CHECK
endif

    CHECK
    if count = ringsize then
        if min = id(x) then
            become LEADER
        else
            become FOLLOWER
        endif
    endif

```

Figure 86: All the way 2

Per quest'ultima assumiamo che tutti i canali di comunicazione implementino una politica **FIFO** e che dunque i messaggi vengano inoltrati da ogni entità nello stesso ordine in cui sono ricevuti. Tuttavia questa assunzione non fa parte del set base di assunzioni per la leader election e dunque non possiamo far terminare localmente il protocollo nel caso in cui un'entità riceva indietro il proprio valore. Nella nostra configurazione, invece, un'entità *conta quanti*

*valori diversi riceve e quando il counter uguale a n termina la propria esecuzione.* Un ulteriore problema che le entit non conoscono il numero n di nodi facenti parte dell'anello. La soluzione quella di utilizzare un secondo counter, stavolta contenuto in ogni messaggio, incrementato da ogni altra entit che lo riceve: in questo modo quando x riceve indietro il proprio messaggio conosce anche il numero di nodi della rete. Quanto detto sull'algoritmo valido anche nel caso di considerare link bidirezionali. La complessit dell'algoritmo riassunta brevemente in figura.

Each identity crosses each link -->  $n^2$   
 The size of each message is  $\log(id + counter)$   
 $O(n^2)$  messages  
 $O(n^2 \log (MaxMsg))$  bits

Figure 87: All the way: complessit

#### 4.6.5 AsFar (as it can)

Le assunzioni per questo protocollo sono: **Unidirectional/Bidirectional link, Different ids, Local Orientation.**

L'idea molto semplice: **non necessario ricevere e inoltrare messaggi che hanno un ID pi grande degli id che sono stati visti.**

#### 4.6.6 AsFar: terminazione

Sappiamo che il messaggio con l'ID pi piccolo sar sempre inoltrato dagli altri nodi, ritornando al mittente originario. Inoltre sappiamo che ogni altro messaggio (con ID pi grande) sar prima o poi terminato da un'altro nodo della rete. Questi due caratteristiche del protocollo ci forniscono un meccanismo di terminazione. Tuttavia solo l'entit che vede tornare indietro un messaggio con il proprio ID sa di essere il leader e dunque dovr comunicarlo alle altre entit della topologia.

#### 4.6.7 AsFar: complessit

Rispetto ad all the way il protocollo impiega un numero minore di messaggi. Il percorso di un messaggio all'interno della topologia viene interrotto nel

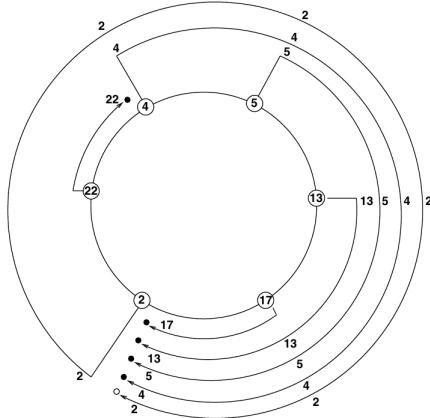


Figure 88: AsFar

```

States: S={ASLEEP, AWAKE, FOLLOWER, LEADER}
S_INIT={ASLEEP}
S_TERM={FOLLOWER, LEADER}           --- unidirectional version

ASLEEP
Spontaneously
    send("Election",id(x)) to right
    min:= id(x)
    become AWAKE

Receiving("Election", value)
    send("Election",id(x)) to right
    min:= id(x)
    If value < min then
        send("Election", value) to other /* this could be
                                           avoided if
                                           id(x)>value
    endif
    become AWAKE

```

Figure 89: AsFar: algoritmo 1

caso in cui si trovi un id più piccolo, dunque il numero totale di messaggi dipende da come gli ID sono dislocati nell'anello.

### Caso peggiore

Il caso peggiore risulta quello nel quale gli ID sono dislocati in ordine crescente e i messaggi sono inviati nella direzione "crescente". In questo caso non è importante il valore reale di un id ma solo se è più piccolo o più grande rispetto agli altri. Quello che è importante dunque è il **rank (i)** degli ID.

- Nel caso  $i = 1$  (rango più piccolo) abbiamo che il messaggio si propaga nella rete fino a tornare al mittente impiegando  $n$  messaggi.

```

AWAKE

Receiving("Election", value)
    if value < min then
        send("Election", value) to other
        min := value
    else
        If value = id(x) then NOTIFY endif
    endif

Receiving(Notify)
    send(Notify) to other
    become FOLLOWER

```

```

NOTIFY
    send(Notify) to right
    become LEADER

```

Figure 90: AsFar. algoritmo 2

- Nel caso  $i = 2$  il messaggio viene fermato solo dall'entità che ha rango  $i = 1$  impiegando  $(n - 1)$  messaggi.
- In generale nel caso  $i + 1$  si impiegano  $(n - i)$  messaggi

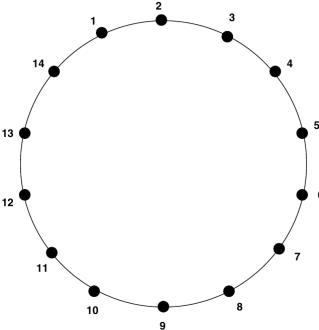


Figure 91: AsFar: complessit caso pessimo

Abbiamo diminuito il numero di messaggi di almeno metà. Dal punto di vista teorico non è un grande risultato ma dal punto di vista pratico può essere considerato soddisfacente. Inoltre abbiamo per ora analizzato solo il caso pessimo.

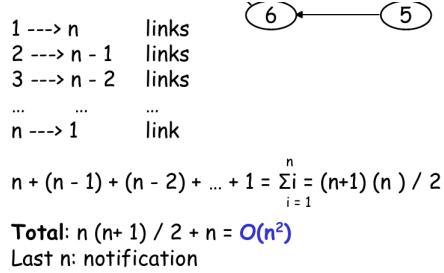


Figure 92: AsFar: complessit caso pessimo

**Caso migliore** La dislocazione degli id quella del caso pessimo ma i messaggi sono inviati nella direzione decrescente. Complessit in figura. Il caso medio ha complessit  $O(n \log(n))$

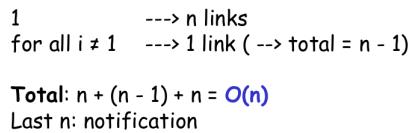


Figure 93: AsFar: complessit caso migliore

#### 4.6.8 Controlled Distances

Le assunzioni per il protocollo sono: **Bidirectional ring, different ids, Local orientation.** L'intento quello di ottenere un protocollo per la leader election che **garantisca**  $O(n \log(n))$  per la complessit in messaggi. Vediamo le idee che stanno alla base del protocollo:

- **Distanza limitata:** ogni entità impone un limite alla distanza percorsa da ogni messaggio.
- **Messaggi di feedback:** se durante il percorso il messaggio non terminato da nessuna entità con id minore allora, torna indietro al mittente originario per ottenere autorizzazione ad andare più avanti.
- **Controllare entrambi i lati:** l'operazione di cui sopra viene fatta inviando un messaggio in entrambe le direzioni (con lo stesso limite); solo se entrambi ritornano al mittente si permette al messaggio di

viaggiare per una distanza maggiore. Se un messaggio torna indietro dal lato opposto allora il mittente diviene leader.

Considerando questi primi tre punti si evince che ogni entità tenta più volte di farsi eleggere leader (aggiornando la distanza percorribile dal messaggio). Un tentativo chiamato **Electoral Stage**: un'entità passa allo stage successivo solo se sopravvive (riceve indietro i messaggi). Se almeno uno dei due messaggi non tornato al mittente, quest'ultimo viene **sconfitto** (defeated) e si occuper solamente del forwarding dei messaggi provenienti dalle altre entità (se riceve un messaggio di terminazione allora termina). **Importante** sottolineare come l'incertezza riguardo ai delay di consegna dei messaggi introduca la possibilità che le entità siano nello stesso momento in stage elettorali diversi. Il problema nel concreto quello di stabilire quanto un'entità debba aspettare il ritorno di entrambi i messaggi prima di dichiararsi sconfitta (i delay sono limitati ma non noti).

- **L'ID più piccolo vince:** se in qualsiasi momento un'entità riceve un messaggio con un Id più piccolo allora viene sconfitto (a prescindere dallo stage in cui si trova)

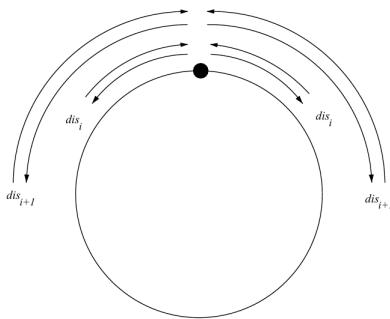


Figure 94: Distanza controllata

#### 4.6.9 Controlled Distances: correttezza e costo

Per quanto riguarda la correttezza sappiamo che al termine del protocollo avremo un solo leader poiché:

- L'id più piccolo attraversa tutta la topologia;

```

States: S={ASLEEP, CANDIDATE, DEFEATED, FOLLOWER, LEADER}
S_INIT={ASLEEP}
S_TERM={FOLLOWER, LEADER}

```

```

ASLEEP
Spontaneously
    INITIALIZE
        become CANDIDATE

    Receiving("Forth", id*, stage*, limit*)
        if id* < id(x) then
            PROCESS_MESSAGE
            become DEFEATED
        else
            INITIALIZE
            become CANDIDATE

```

Figure 95: Distanza controllata 1

```

CANDIDATE

    Receiving("Forth", id*, stage*, limit*)
        if id* < id(x) then
            PROCESS_MESSAGE
            become DEFEATED
        else
            if id* = id(x) then NOTIFY

    Receiving("Back", id*)
        if id* = id(x) then CHECK

    Receiving("Notify")
        send("Notify") to other
        become FOLLOWER

DEFEATED

    Receiving(*)
        send(*) to other
        if * = Notify then
            become FOLLOWER

```

Figure 96: Distanza controllata 2

- la distanza percorribile dai messaggi aumentata in senso **strettamente monotono**.
- un'entità prima o poi riceve un messaggio dalla parte opposta rispetto a quella di invio.
- non abbiamo bisogno dell'ordinamento per i messaggi (i messaggi possono essere di stage differenti)

```

INITIALIZE
    stage := 1
    limit := dis(stage)
    count := 0
    send("Forth", id(x), stage, limit) to N(x)

PROCESS-MESSAGE
    limit* := limit* - 1
    if limit* = 0 then
        send("Back", id*, stage*) to sender
    else
        send("Forth", id*, stage*, limit*) to other

CHECK
    count := count + 1
    if count = 1 then
        count := 0
        stage := stage + 1
        limit := dis(stage)
        send("Forth", id(x), stage, limit) to N(x)

NOTIFY
    send("Notify") to right
    become LEADER

```

Figure 97: Distanza controllata 3

- **Messaggi:** se la distanza raddoppiata ad ogni stage

$$dis(i) = 2^{i-1}$$

allora la complessità  $O(n \log(n))$

- **Tempo:**  $2dis(i)$  tempo necessario al messaggio con l'id più piccolo per raggiungere di nuovo il mittente. Per la notifica finale il tempo necessario  $2n$ .

- in totale

$$2n + \sum_{i=1} 2dis(i) = O(n)$$

#### 4.6.10 Stage

L'idea introdotta con gli electoral stage prevede che un'entità faccia più tentativi successivi prima di essere eletto leader. In questo modo è possibile implementare algoritmi che non richiedono comunicazioni sincrone tra le entità della topologia.

In questa sezione si mostrerà come è possibile avere un algoritmo di leader election ancora più performante, **senza l'utilizzo delle distanze controllate e**

**senza il bisogno di messaggio di feedback.** Per semplicità introduciamo l'assunzione di **message ordering** che sarà eliminata in seguito.

1. Un candidato  $x$  invia un messaggio contenente il proprio ID in entrambe le direzioni. Questo messaggio continua il suo percorso finché non incontra un altro candidato.
2. Per simmetria  $x$  riceve due messaggi, uno da destra e uno da sinistra; verrà sconfitto se uno di questi contiene un ID più piccolo del suo. Se invece i messaggi ricevuti hanno ID più alto, l'entità può iniziare lo stage successivo. Se entrambi i messaggi contengono il proprio ID allora diventa leader e notifica agli altri nodi questa informazione.
3. un nodo sconfitto inoltra i messaggi dei nodi ancora attivi ed ogni nodo non iniziatore viene sconfitto se riceve un messaggio di election.

#### 4.6.11 Stage: correttezza e complessità

Per quanto riguarda la correttezza è facile osservare che ad ogni stage elettorale il numero di entità candidate diminuisce. Infatti l'entità con ID minimo non viene mai sconfitta e al contempo elimina le altre entità finché non riceve indietro i suoi messaggi. In questo modo il leader sa che tutte le altre entità sono state sconfitte e che può notificare di essere il leader.

Dal punto di vista della complessità in messaggi abbiamo che:

- Ad ogni stage ogni candidato invia od inoltra 2 messaggi dunque il totale sarà  $2n$ . I bit per messaggio sono  $2n * (\log(n))$
- Ad ogni passo almeno la metà delle entità viene eliminata.

$$n_0 = n$$

$$n_1 = n/2$$

$$n_i = n/2^i$$

$$n/2^k = \text{con}k = \log(n)$$

**Totale:**

$$2n * (\log(n)) + 3n = O(n \log(n))$$

*PROTOCOL Stages.*

- States:  $\mathcal{S} = \{\text{ASLEEP}, \text{CANDIDATE}, \text{WAITING}, \text{DEFEATED}, \text{FOLLOWER}, \text{LEADER}\}$
- $S_{\text{INIT}} = \{\text{ASLEEP}\}; S_{\text{TERM}} = \{\text{FOLLOWER}, \text{LEADER}\}$ .
- Restrictions:  $\text{IR} \cup \text{Ring}$ .

```

ASLEEP
    Spontaneously
    begin
        INITIALIZE;
        become CANDIDATE;
    end

    Receiving ("Election", id*, stage*)
    begin
        INITIALIZE;
        min:= Min(id*,min);
        close (sender);
        become WAITING;
    end

CANDIDATE
    Receiving ("Election", id*, stage*)
    begin
        if id* ≠ id(x) then
            min:= Min(id*,min);
            close (sender);
            become WAITING;
        else
            send(Notify) to N(x);
            become LEADER;
        end
    end

WAITING
    Receiving ("Election", id*, stage*)
    open (other);
    stage:= stage+1;
    min:= Min(id*,min);
    if min= id(x) then
        send("Election", id(x), stage) to N(x);
        become CANDIDATE;
    else
        become DEFEATED;
    endif
    end

DEFEATED
    Receiving (*)
    begin
        send (*) to other;
        if * = Notify then become FOLLOWER endif;
    end

```

Figure 98: Algoritmo stages

#### 4.6.12 Stages: rimuovere message ordering

L'assunzione di message ordering assicura che i messaggi ricevuti da un candidato nello stage i siano originati da candidati nello stesso stage i. Possiamo eliminare questa assunzione modificando il protocollo **Controlled Dis-**

tances in modo tale che vengano forzati i suoi effetti.

- Ogni messaggio contiene anche il numero di stage nel quale originato.
- Ogni messaggio contenente numero di stage  $j$  una volta consegnato viene processato solo dopo quelli degli stage  $i, \dots, j - 1$ ;
- Ogni qualvolta un candidato  $x$  riceve un messaggio di elezione contenente  $ID(y)$  o  $ID(z)$ , rispettivamente da  $y = r(i, x)$  e  $z = l(i, x)$  invier un feedback positivo a  $y$  se  $ID(y) < \min(ID(x), ID(z))$ , se  $ID(z) < \min(ID(x), ID(y))$  lo invier a  $z$ . Se  $ID(x)$  il minimo tra i tre non si invia nessun feedback.
- Un candidato sopravvive e pu iniziare il prossimo stage elettorale solo se riceve feedback positivi da entrambi i lati dell'anello. Un nodo che invia messaggi di feedback sa già che non passer al prossimo stage elettorale.
- I nodi che invece sono stati sconfitti attendono dei messaggi di feedback che sono destinati a non arrivare. Come pu quindi una di queste entità sapere che stata sconfitta? Si utilizzano i messaggi degli step successivi come feedback negativo: **Un'entità che aspetta un feedback per lo stage  $i$ -esimo, diviene sconfitta se riceve un messaggio (con id sicuramente più piccolo) da uno stage  $i+1$ .**

#### 4.6.13 Stages: correttezza terminazione

Consideriamo un'entità  $x$  che da ID minimo e che sarà quindi eletta leader:

- non invia mai feedback positivi e riceve sempre due feed positivi.
- Questo significa che ogni suo vicino, anch'esso candidato non sopravvive allo stage.
- Il numero di candidati per ogni stage monotonica decrescente.

Vediamo adesso la complessità del protocollo:

- Se  $x$  sopravvive allora deve aver ricevuto un feedback da  $r(i, x)$  e  $l(i, x)$  che pertanto sono eliminati. Inoltre  $r(i, x)$  e  $z = l(i, x)$  non inviano almeno uno dei feedback ai loro vicini poiché lo inviano ad  $x$ : dunque anche i vicini  $r^2(i, x)$  e  $l^2(i, x)$ .

- **numero di stage:**  $n_{i+1} \leq n_i/3 \rightarrow \log_3 n$
- Per quanto riguarda i **messaggi** abbiamo:
  - ogni candidato invia un messaggio per link dunque abbiamo  $2n$  messaggi iniziali;
  - ogni entità invia un messaggio di feedback dunque questi sono  $n$  ad ogni stage.
  - Il totale dei messaggi per stage  $3n$
- Il totale dei messaggi  $3n \log_3 n + O(n) \leq 1.89 n \log(n) + O(n)$

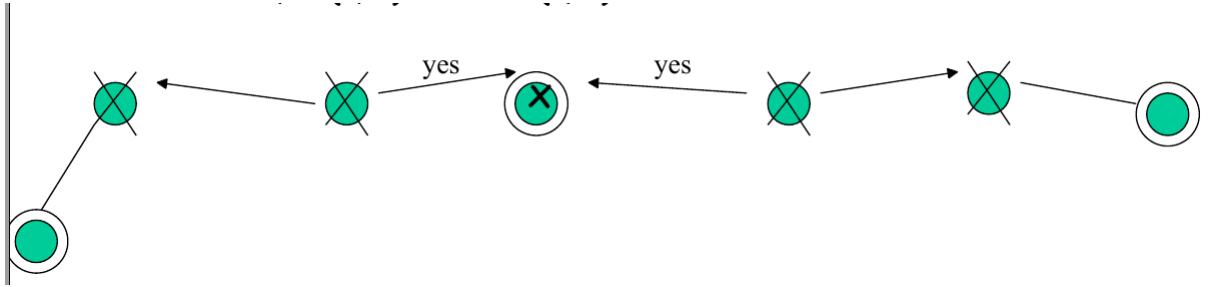


Figure 99: Control, sopravvissuti ad ogni stage

#### 4.6.14 Alternating Steps

Assunzioni per il protocollo: **Different ids**, **Bidirectional link and sense of direction**, local orientation, Message ordering.

Il protocollo rappresenta una versione "rallentata" di stages.

- Ogni entità invia un messaggio alla sua destra contenente il proprio id.
- Ogni entità compara l'id ricevuto dalla sua sinistra con il proprio: diviene passiva se l'id ricevuto è più piccolo.
- tutte le unità che rimangono attive allo stage successivo inviano un messaggio alla loro sinistra con confronti analoghi a quelli visti nello step precedente: si ripetono questi passaggi fino all'elezione del leader (l'entità riceve indietro i propri messaggi). Complessità in figura.

Per quanto riguarda la **correttezza** abbiamo le consuete affermazioni: l'entità con l'id più basso non sarà mai eliminata mentre quella con id più alti saranno eliminati ad ogni passo. Le entità decrescono in senso monotono ad ogni round.

### Complessità

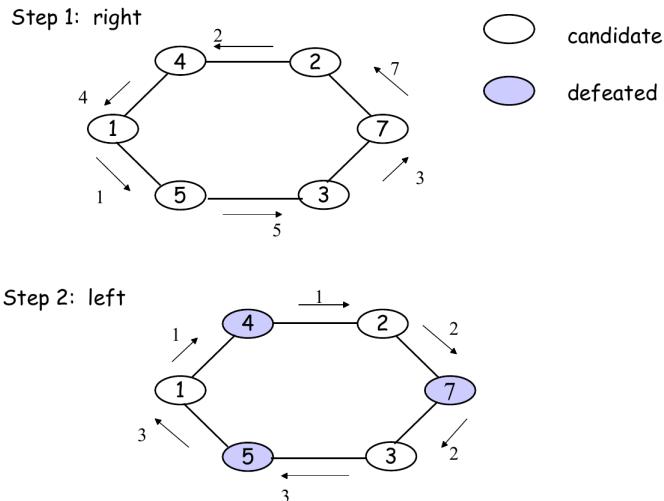


Figure 100: Alternating steps

States:  $S = \{\text{ASLEEP}, \text{CANDIDATE}, \text{WAITING}, \text{DEFEATED}, \text{FOLLOWER}, \text{LEADER}\}$   
 $S_{\text{INIT}} = \{\text{ASLEEP}\}$   
 $S_{\text{TERM}} = \{\text{FOLLOWER}, \text{LEADER}\}$   
Restrictions: **IR**; Oriented Ring; Message Ordering

**ASLEEP**  
*Spontaneously*  
**INITIALIZE**  
**become CANDIDATE**

*Receiving("Election", id\*, step\*)*  
**INITIALIZE**  
**PROCESS\_MESSAGE**  
**become CANDIDATE**

Figure 101: Alternating steps algoritmo 1

```

CANDIDATE
Receiving("Election", id*, step*)
  if id* <> id(x) then
    PROCESS_MESSAGE
  else
    send(Notify) to N(x)
    become LEADER

DEFEATED
Receiving(*)
  send(*) to other
  if * = Notify then
    become FOLLOWER

INITIALIZE
step := step + 1
min := id(x)
send("Election", id(x), step) to right
close-port(right)

PROCESS_MESSAGE
  if id* < min then
    open(other)
    become DEFEATED
  else
    step := step + 1
    send("Election", id(x), step) to
    sender
    close-port(sender)
    open(other)

```

Figure 102: Alternating steps algoritmo 2

Analyze # of steps in worst case:

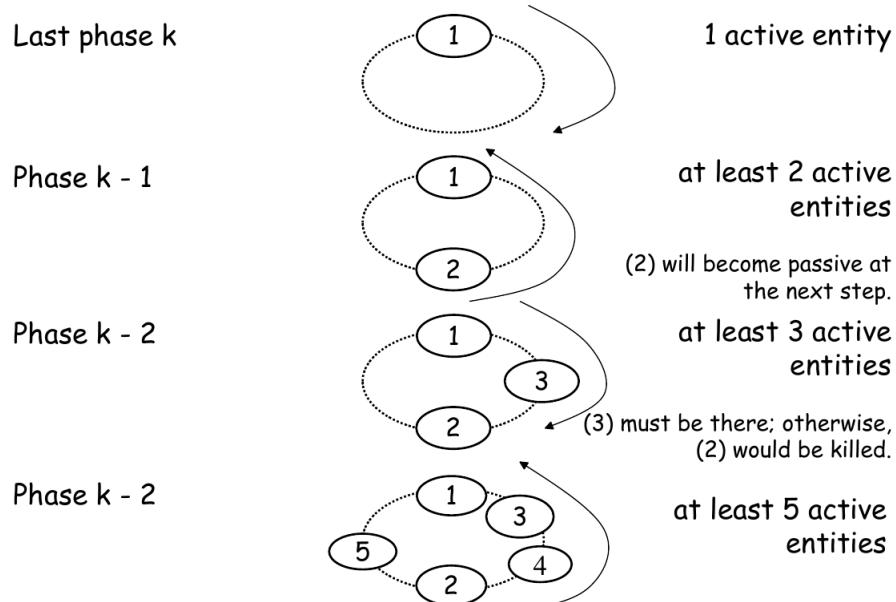


Figure 103: Alternating steps complessit

**# steps** = index of the lowest Fibonacci number  $\geq n$

$F_1$	= 1
$F_2$	= 2
$F_3$	= 3
$F_4$	= 5
$F_5$	= 8
...	

$F_k$	= i = ?
= approx. $1.45 \log_2 n$	

**# Messages** = n for each step

**Total** = approx.  $1.45 n \log_2 n$

Figure 104: Alternating steps complessità 2

#### 4.6.15 Unidirectional o Bidirectional links?

Ora che abbiamo descritto i principali algoritmi per la leader election in un anello, ci domandiamo se l'assunzione **Bidirectional link** necessaria per tutti i protocolli.

- per quanto riguarda **All the way**, e **AsFar** entrambi i protocolli funzionano in tutti e due i casi.
- per quanto riguarda **Distances**, **Stages**, **Stages with feedback**, **Alternate** utilizzano link bidirezionali (per esempio per ottenere dei feedback). Grazie a questa assunzione **abbiamo ridotto il costo computazionale da  $O(n^2)$  a  $O(n \log(n))$**

**La domanda** : *davvero necessaria l'assunzione bidirectional link?* In caso positivo dovremmo, al momento dell'implementazione, investire in ulteriore hardware per le comunicazioni (full duplex lines).

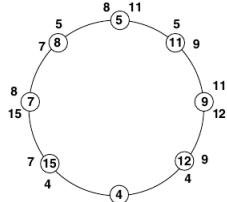
**La risposta** NO! Facciamo vedere come possibile simulare uno di questi algoritmi (Stages, sul libro si trovano tutti) utilizzando link unidirezionali.

##### Unidirectional Stages

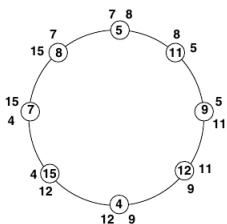
Dividiamo l'operazione *invia in entrambe le direzioni* in due sotto operazioni:

- Invio il valore in una direzione

- Invio nella stessa direzione il valore che ricevo dalla direzione opposta



(b)



(d)

Figure 105: Alternating steps complessit 2

Nella versioni bidirezionale e unidirezionale le entità hanno dopo il primo passo gli stessi valori con cui prendere una decisione **solo che queste sono spostate sul candidato successivo**

## 4.7 Yo-Yo: algoritmo distribuito per la ricerca del minimo

### 4.7.1 Struttura

L'algoritmo consiste di due parti:

- **Fase preliminare (pre-processing phase):** Ogni entità  $x$  scambia il proprio identificativo con i propri vicini. Alla fine di questa fase ogni entità conosce gli ID dei suoi vicini. Successivamente  $x$  orienta gli archi nella direzione dell'entità con l'ID maggiore.  
**Proprietà:** il grafo così ottenuto è un **DAG** (directed acyclic graph: nozioni base in figura).

**Prova (per assurdo):** assumiamo che esista un ciclo  $x_0, \dots, x_k$  ci significa che  $id(x_0) < id(x_1) < \dots < id(x_k)$ . Assurdo poiché dovrebbe essere  $id(x_{k-1}) < id(x_k) = id(x_0)$ .

- **Sequenza di iterazioni (sequence of iteration):** ogni iterazione coincide con uno stage elettorale; dopo ogni stage le entità candidate decrescono in senso monotono. Ogni iterazione consta di due parti:
  1. Il **primo step** inizializzato dalle **sources** che inviano il proprio Id a tutti i vicini.
  2. Un nodo interno attende di ricevere gli ID da tutti gli archi entranti, **computa il minimo e lo invia attraverso tutti gli archi uscenti**.
  3. Un sink aspetta di ricevere valori da tutti i propri vicini, **computa il minimo e comincia il secondo step**.

1. Il **Secondo step** inizializzato dai **sink**: ognuno di questi risponde **YES** a tutti i vicini che hanno inviato il valore più piccolo.
2. Un **nodo interno** attende finché non riceve un voto da tutti i suoi out neighbors, se **tutti i voti sono YES invia YES altrimenti NO**.
3. Una **source** aspetta finché non riceve voti da tutti i suoi vicini uscenti, se tutti i voti sono YES, **sopravvive** e va all'iterazione successiva.

Prima di cominciare l'iterazione successiva dobbiamo **modificare** il DAG:

1. Quando un nodo invia un NO ad un suo in-neighbor **inverte la direzione dei link**.
2. Quando un nodo riceve un NO su un link, ruota la direzione di quel link.

Ogni source che riceve un NO smette di essere una source e diventa un sink, alcuni sink diventano nodi interni e alcuni nodi interni diventano sink. **Il numero delle sources decresce ad ogni iterazione**.

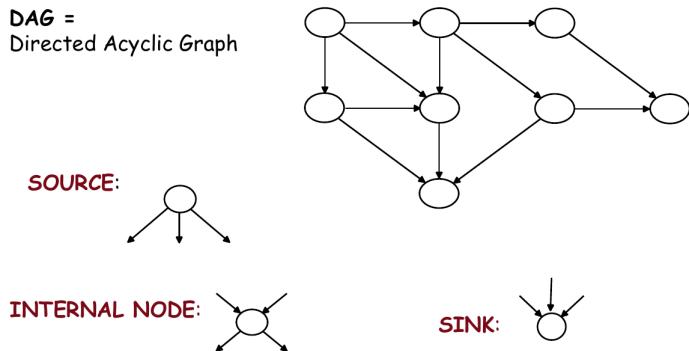


Figure 106: DAG: nozioni base

Given an arbitrary undirected graph, smaller entities are directed towards bigger (by exchanging messages with neighbours)

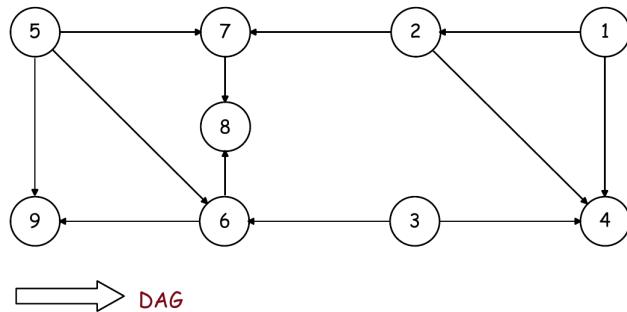


Figure 107: DAG ottenuto con la fase di setup

#### 4.7.2 Terminazione

1. Ad ogni passo almeno una source diventa nodi interno o un sink.
2. Un nodo interno non pu divenire una source dopo la rotazione degli archi.
3. Un sink non pu diventare una source dopo la rotazione degli archi

Segue che **il numero delle sources decresce in senso monotonico**. Tuttavia come pu un nodo venire a conoscenza che il vincitore ? Infatti ricever YES in ogni iterazione, anche se ha sconfitto tutti le altre sources. Una soluzione rappresentata dalla tecnica del **pruning**, che consiste nell'eliminare

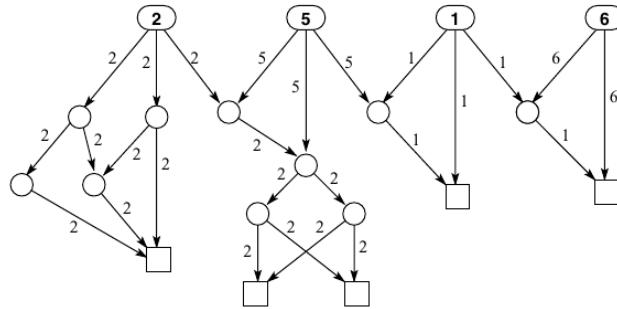


Figure 108: Dalle source ai sink

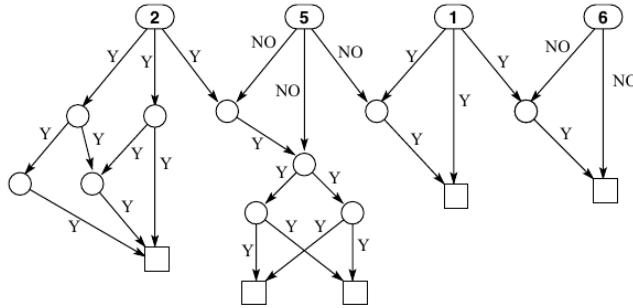


Figure 109: Dai sink alle sources

i nodi e i link che sono inutili; alla fine un solo nodo rimarr vivo ( il vincitore).

### Pruning:

1. Se un sink una foglia allora inutile (il padre potrebbe diventare un sink);
2. Se un nodo riceve, nella prima fase, lo stesso valore da molti in neighbors allora pu chiedere a tutti tranne uno di eliminare i propri links.
3. **Se un dag ha una singola source dopo un'iterazione sar composto da un solo nodo la source stessa: TERMINAZIONE.**

Il pruning avviene nella fase di voto inserendo informazioni all'interno del messaggio.

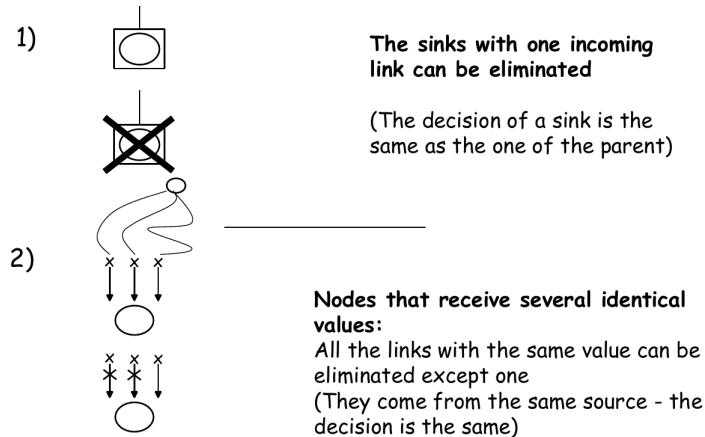


Figure 110: Regole di pruning

#### Without Pruning

There are 2 messages for each link at each phase

- The number of phases is:  $\log(\# \text{ sources})$

$(s = \# \text{ sources})$

TOT:  $m + 2m \log s$

↑  
For initialization

With Pruning: ?

Figure 111: YO-YO complessit

## 4.8 Election in una rete dinamica

### 4.8.1 The Bully Algorithm: introduzione

Prima di tutto vediamo quali sono le **assunzioni** per l'algoritmo:

- Il sistema **sincrono**. Questa risulta essere un'assunzione molto forte rispetto ai clock delle entità coinvolte:
  - **Clock sincronizzati**: tutti i clock sono incrementati simultaneamente.

- **Delay di comunicazione limitati:** esiste un **upper bound** per i delay di comunicazione.

2. le entit **possono fallire** in qualsiasi momento (anche durante l'esecuzione)
3. abbiamo un **failure detector** che individua i nodi falliti. Come possiamo stabilire se un'entit fallita o meno ? Si utilizzano messaggi di **heartbeat**:
  - Un'entit x invia un messaggio di heartbeat a tutti i suoi vicini;
  - se un vicino non risponde entro un intervallo di tempo stabilito (threshold) l'entit considerata fallita. Poich siamo a conoscenza dell'upperbound di consegna dei messaggi la scelta pi logica che il timeout sia un multiplo di questa soglia.
4. Le **comunicazioni sono reliable** (upperbound per il delay di consegna dei messaggi esiste ed finito).
5. La topologia un **grafo completo**: ogni entit conosce il proprio ID e quella delle altre entit, ma non sa quali di queste sono attive o fallite.
6. Iniziatori multipli.

Vediamo quali sono i messaggi utilizzati:

- **Election:** inviato per iniziare/annunciare un'elezione;
- **Answer:** risposta ad un messaggio di election (im alive);
- **Coordinator:** inviato dal leader per notificargli la propria vittoria.

#### 4.8.2 Bully: algoritmo

Un'entit x inizia il protocollo se scopre che:

- Il leader corrente fallito;
- x si riavvia in seguito ad un fallimento.

Gli step dell'algoritmo sono i seguenti:

1. Se x ha l'ID con il valore pi alto allora invia un messaggio di coordinatore a tutte le altre entit nella rete;

2. In caso contrario invia un election message a tutte le entit con un id pi alto del suo;
3. se non riceve risposta diviene leader e lo comunica con messaggi coordinator;
4. se riceve un messaggio da un nodo con un id piu alto, x viene sconfitto ed aspetta un messaggio di notifica.
5. se non riceve un messaggio di notifica x fa ripartire l'algoritmo.
6. Se x riceve un messaggio di election da un'entit con id piu basso, far partire un nuovo processo di elezione del leader, inviando messaggi di election a tutti i nodi con id piu alto.
7. se x riceve un messaggio di notifica tratta il sendere come il coordinatore.

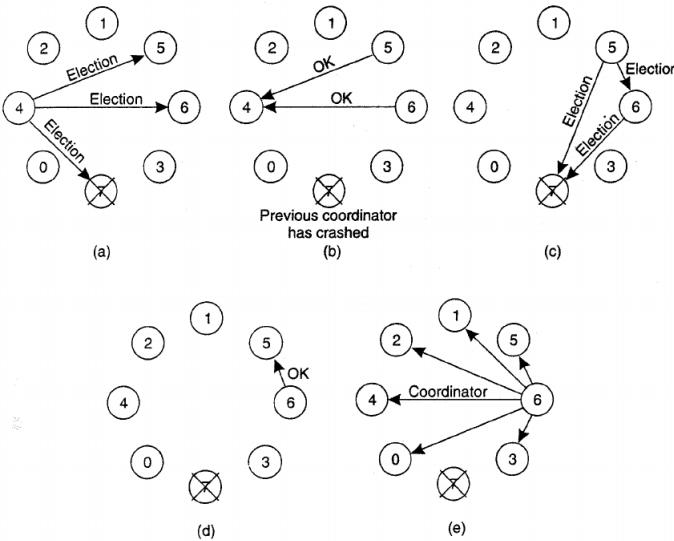


Figure 112: Bully esempio

#### 4.8.3 Elezione in una rete wireless Ad-Hoc

Per prima cosa utile ricordare che in una rete wireless:

## Message Complexity (Worst Case)

The entity with the lowest id starts the election

$$\begin{aligned} \text{Id 1} &\rightarrow n - 1 \text{ mgs} \\ \text{Id 2} &\rightarrow n - 2 \text{ mgs} \\ \dots \\ \text{Id } N-1 &\rightarrow 1 \text{ mgs} \end{aligned} \quad \sum_{i=1}^n i \approx \Theta(n^2)$$

There are also the  $n$  notification messages

Figure 113: Bully complessit

- la consegna dei messaggi **non reliable**;
- La topologia della rete pu variare nel tempo;
- Le entit conoscono solo i loro vicini.

Questo tipo di rete **non si appoggia su un'infrastruttura preesistente** e dunque **ogni nodo partecipa al routing** inoltrando messaggi per altri nodi. Ovviamente i **canali di comunicazione sono stabiliti dinamicamente** sulla base della connettività di rete e grazie all'algoritmo di routing. Una rete di questo tipo ha dei vantaggi e pu essere utilizzata con profitto in relazione al contesto di applicazione:

- Sono altamente performanti;
- Non hanno bisogno di infrastrutture, dunque ha costi ridotti.
- la distribuzione delle informazioni rapida vicino al sender.
- Non esiste un solo punto di fallimento.
- La topologia dinamica poich le entit possono muoversi.

Gli algoritmi per la leader election finora descritti non funzionano con reti che cambiano la loro topologia: abbiamo bisogno di algoritmi specifici per questo tipo di reti.

#### 4.8.4 Rete Ad Hoc: algoritmo leader election

In primo luogo riformuliamo il problema della leader election per questo tipo di rete:

*Data una rete di entità con un valore, dopo un numero finito di cambiamenti nella topologia di rete, ogni componente connessa sceglierà prima o poi un leader unico.*

Vediamo adesso le assunzioni iniziali per questo protocollo.

- Ogni entità ha un **valore** (livello di batteria, capacità computazionali ecc);
- Ogni entità ha un **ID unico**. Gli ID sono **ordinati** tra di loro.
- I link sono **bidirezionali** e implementano una politica **FIFO**.
- Un nodo si può spostare, fallire, riavviare la propria esecuzione.
- Ci possono essere **iniziatori multipli**.

Vediamo come avviene l'avvio del protocollo:

1. Il leader di una componente connessa invia periodicamente messaggi di **heartbeat**;
2. Se un nodo non riceve messaggi di heartbeat per un periodo di tempo prestabilito avvia il protocollo di election (più nodi possono avviare la procedura)

Vediamo lo schema base per adesso nel caso di un **iniziatore singolo e senza la possibilità di incorrere in fallimenti**.

1. Costruzione di uno **spanning tree** da parte dell'iniziatore;
2. Le foglie lanciano un **convergecast** con l'obiettivo di stabilire il candidato migliore.
3. La radice dello spanning tree notifica l'identità del leader alle altre entità.
4. Le tipologie di messaggi utilizzate sono: **Election** (inizializza il processo), **Ack** (faccio parte dello spanning tree; questi messaggi contengono anche informazioni riguardo ai candidati), **Leader** (utilizzati per notificare l'identità del leader).

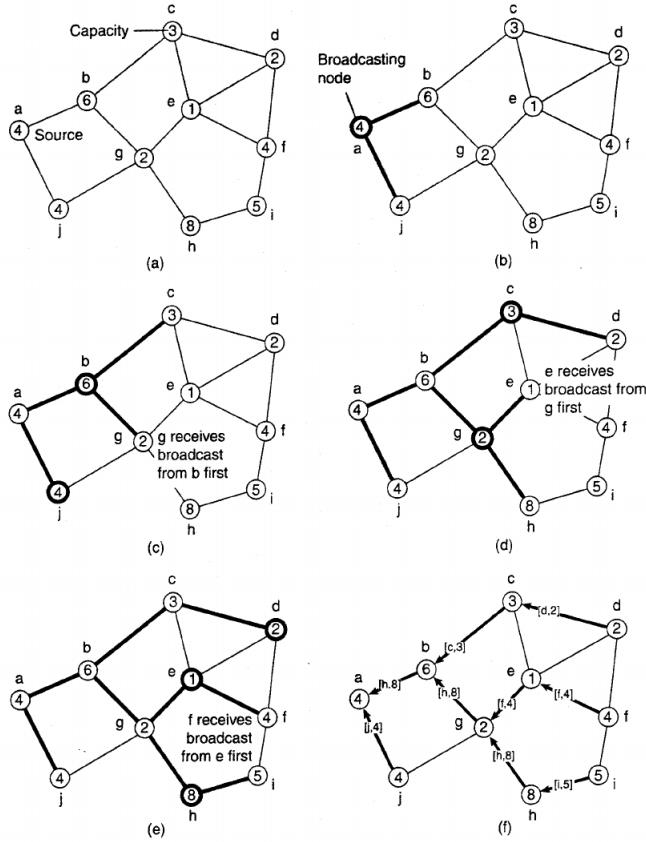


Figure 114: Leader election per reti ad hoc

### PROBLEMA 1:

Come possiamo introdurre la possibilità di gestire **iniziatori multipli** e dunque **computazioni concorrenti**?

- stabiliamo che un nodo possa partecipare ad un solo processo di election;
- Si utilizzano degli indici (totalmente ordinati) per identificare le computazioni;
- Quando un nodo  $x$  riceve un messaggio appartenente ad una computazione con un indice più grande,  $x$  interrompe la propria computazione e si unisce a quella con indice maggiore.

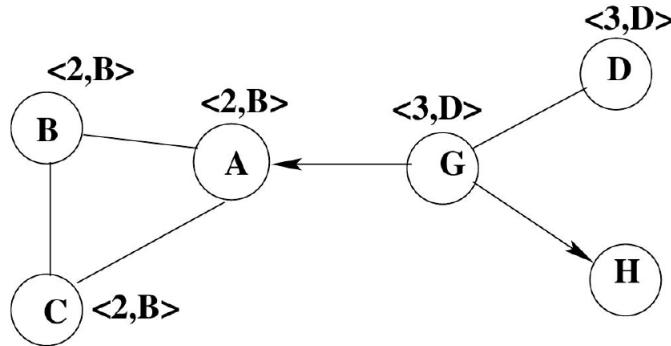


Figure 115: Computazioni concorrenti 1

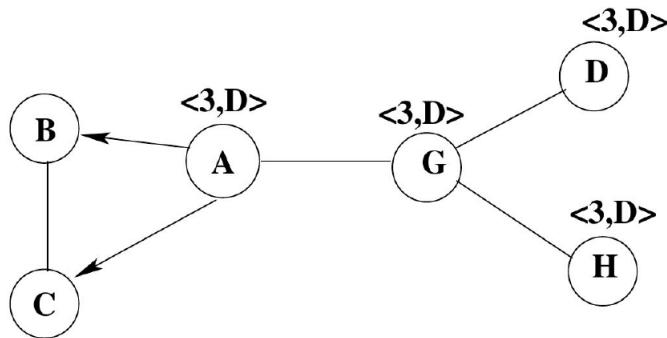


Figure 116: Computazioni concorrenti 2

### PROBLEMA 2:

Dal momento che siamo all'interno di uno spanning tree, una generica entità x deve attendere gli ack provenienti dai propri figli prima di poter inviare un ack al proprio padre: **cosa succede se uno dei figli fallisce?**

1. Se x rileva che y è disconnesso, x smette di attendere un ack e ignora y;
2. Se y non può gestire i propri ack termina il processo agendo come radice dell'albero e notificando il leader

### PROBLEMA 3

**Come possiamo rilevare la disconnessione di uno dei nodi?** Vengono introdotti 2 nuovi tipi di messaggio.

- Ogni nodo nello spanning tree invia messaggi periodici di **probe** a tutti i nodi che attendono un ack.

- Ogni nodo che riceve un probe risponde con un messaggio di **reply**;
- Se non arrivano reply il figlio considerato disconnesso

## PROBLEMA 4

**La mobilit delle entit pu portare ad un merge di partizioni.**

1. Il leader con l'id pi alto vince e diviene leader della partizione ottenuta dal merge delle due.
2. Se una componente ha un leader mentre l'altra sta ancora computando l'algoritmo di elezione del leader, si attende la conclusione del processo e si sceglie il leader migliore come da punto precedente.

## 4.9 Sincronizzazione

### 4.9.1 Introduzione

Nel modello di un sistema distribuito sappiamo che **ogni entit ha il proprio clock** e che **non esiste** una nozione di **tempo globale**. In questo tipo di scenario possiamo incorrere in situazioni nelle quali ad un evento avvenuto dopo un altro assegnato invece un valore temporale precedente. La sin-

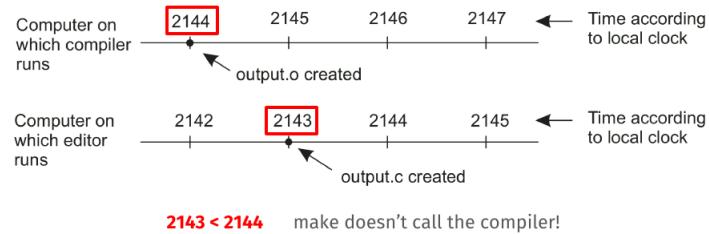


Figure 117: Esempio timestamp erronei 1

cronizzazione dei clock fondamentale in applicazioni che hanno bisogno di una nozione del tempo distribuita e precisa (Sistemi safety critical, sistemi che devono rispettare delle deadline).

Prima di scendere nei dettagli vediamo il funzionamento di un **clock fisico**:

1. sono apparecchi elettronici che contano le oscillazioni che si misurano in un cristallo ad una frequenza stabilita;

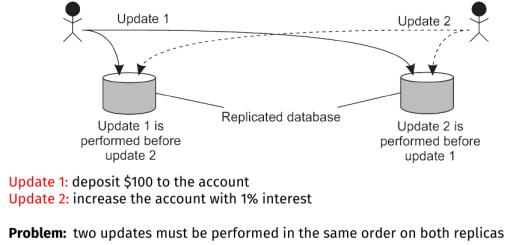


Figure 118: Esempio timestamp erronei 2

- Ad un cristallo sono associati due registri: **counter** e **holding register**. Il primo viene decrementato ad ogni oscillazione, quando raggiunge lo zero viene attivato un interrupt (**clock tick**). Successivamente si ripristina il counter grazie all'holding register.

Nel caso di una singola macchina, se un **processo A** vuole conoscere il tempo corrente **Ta** esegue una system call al sistema operativo. Successivamente un processo **B** pu fare lo stesso ottenendo **Tb**. In questo caso, essendo il clock lo stesso, ne segue che  $Ta \leq Tb$ .

**PROBLEMA:** Se consideriamo pi macchine, i rispettivi cristalli non eseguono mai alla stessa frequenza: si parla quindi di **clock skew**, si registreranno valori diversi se si leggono simultaneamente clock differenti. **In sistemi distribuiti questo problema amplificato** da possibili e arbitrari delay di consegna per i messaggi o addirittura dalla perdita degli stessi.

#### 4.9.2 Sincronizzazione di clock fisici

Esistono due tipi di sincronizzazione :

- **Esterne:** tutte le macchine sono sincronizzate grazie all'utilizzo di un **clock di riferimento esterno**. Un esempio **UTC**: i clock di riferimento sono trasmessi in broadcast dai satelliti.
- **Interna:** tutte le macchine sono sincronizzate tra di loro.

Vediamo due algoritmi per la sincronizzazione:

**Algoritmo di Christian** (Sincronizzazione esterna):

1. Un client invia una richiesta al server inviando un timestamp **T0**.
2. Il server risponde inviando il suo tempo **T**.

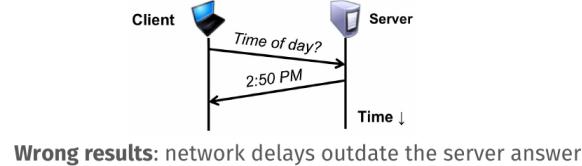


Figure 119: Problemi di sincronizzazione in contesto distribuito

3. Il client associa un timestamp quando riceve il messaggio dal server. In questo modo il client pu stimare **round trip time**:

$$(T_1 - T_0)/2$$

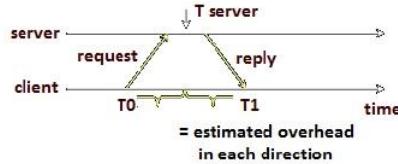


Figure 120: Algoritmo di Christian

**Algoritmo di Berkeley** (Sincronizzazione interna): l'idea quella di avere un server che scansiona periodicamente tutte le macchine, calcola una media dei clock e informa le altre affinch modifichino il proprio in relazione a questa informazione. Le **assunzioni** sono:

1. Non esiste un clock esterno di riferimento;
2. Ogni macchina ha un clock locale accurato;

Vediamo l'algoritmo:

1. Si scegli un leader attraverso un algoritmo di election;
2. il leader interroga gli altri nodi che rispondo con il proprio timestamp;
3. calcola il **RTT** e stima il valore temporale suo e delle altre entit e ne **calcola la media**.
4. Infine il leader invia ad ogni entit un valore (positivo o negativo) da sommare ai propri clock.

### 4.9.3 Network Time Protocol (NTP)

Questo protocollo permette ad un client di sincronizzare il proprio clock ad UTC. Il protocollo più utilizzato nella rete internet e rappresenta lo stato dell'arte per i protocolli di sincronizzazione di reti non affidabili. È stato specificamente ideato per superare i problemi dovuti alla connettività della rete.

In primo luogo è importante specificare che le entità partecipanti (server e time sources) sono organizzate in una gerarchia che vede alla sua estremità macchine con clock ad alta precisione (atomic clock) e, man mano che si scende di livello, macchine meno precise (noioso, prosegue in figura).

Server and time sources are arranged in layers (strata)

- Stratum 0 => high-precision timekeeping devices such as atomic clocks (reference clocks)
- Stratum 1 => machines whose system time is synchronized to within a few microseconds of their attached stratum 0 devices
- ....

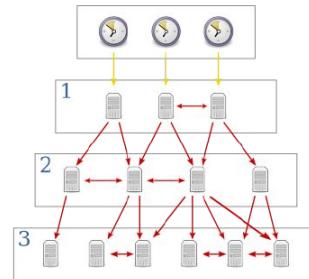


Figure 121: Algoritmo di Christian

### How Does The Protocol Work?

1. The client sends a request timestamped with **T<sub>1</sub>**
2. The server timestamp the receipt of the message with **T<sub>2</sub>**
3. The server replies with a timestamp **T<sub>3</sub>**
4. The client computes the timestamp **T<sub>4</sub>** when the answer arrives

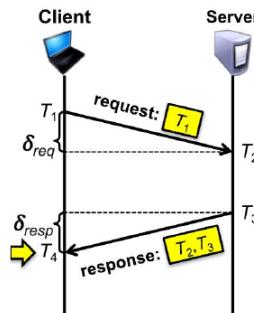


Figure 122: Algoritmo di Christian

## How Does The Protocol Work?

The client computes the offset and the RRT

$$\Theta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

$$\delta = (T_4 - T_1) - (T_3 - T_2)$$

It takes many measurements from different servers and then takes the one with the minimum value of  $\delta$

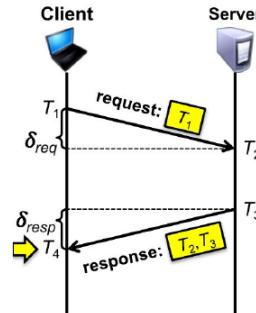
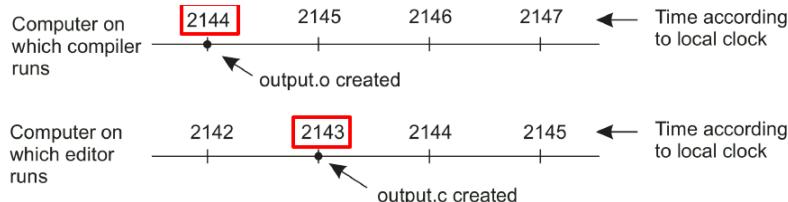


Figure 123: Algoritmo di Christian

### 4.9.4 Clock logici

I clock logici si basano su un'osservazione: **Il tempo reale degli eventi non ha importanza se riusciamo a stabilire una relazione logica tra due eventi.** In poche parole si considera l'ordine di occorrenza di due eventi



We only need to know that **output.c was modified after output.o creation!**

Figure 124: Relazione logica tra due eventi

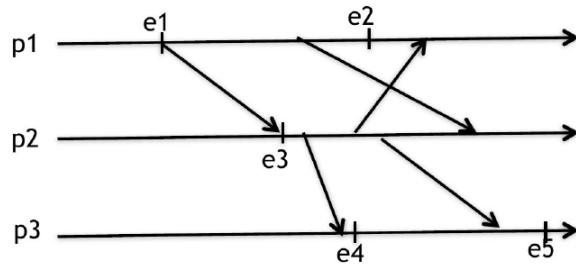
piuttosto che il loro timestamp. Definiamo una relazione di **ordine parziale stretta** (strict partial order)  $<$  definita su un insieme  $E$  come una **relazione binaria**  $<\subseteq E \times E$  che per ogni  $a, b, c \in E$ :

- not  $a < a$  (**irreflexivity**);
- $a < b \wedge b < c \rightarrow a < c$  (**transitivity**);
- se  $a < b$  allora not  $b < a$  (**asymmetry**)

Definita questa relazione di ordine parziale stretto, la **relazione avvenuto prima**  $\rightarrow \subseteq ExE$ , dato un insieme di eventi **E**, il minor ordine parziale sugli eventi tale che:

1. Se gli eventi  $a$  e  $b$  sono avvenuti sulla stessa macchina allora  $a \rightarrow b$  se il timestamp di  $a$  ha un valore precedente a quello di  $b$ .
2. Se gli eventi  $a$  e  $b$  sono avvenuti su macchine diverse,  $a \rightarrow b$  se ha l'evento "invio del messaggio" e  $b$  l'evento "ricezione del messaggio".
3. se  $a \rightarrow b$  e  $b \rightarrow a$  non sono veri  $a$  e  $b$  si dicono **concorrenti** e non possiamo dire niente sul loro ordinamento.

La relazione ordina eventi di macchine che scambiano tra di loro messaggi, se ci non avviene la relazione inutilizzabile.



$$e1 \rightarrow e3, e1 \rightarrow e2, e3 \rightarrow e4, e1 \rightarrow e5, e2 \parallel e5$$

Figure 125: Relazione logica tra due eventi

Possiamo creare una nozione di tempo C condivisa tra tutte le entità di un sistema distribuito sfruttando i clock logici? Nella pratica quello che vogliamo assegnare un valore  $C(e)$  ad un generico evento  $e$ , valore sul quale tutte le entità della topologia concordano. I requisiti da rispettare sono:

- se  $a \rightarrow b \Rightarrow C(a) < C(b)$
- Evitare di generare ulteriori messaggi.

L'idea quella di equipaggiare ogni entità con un clock virtuale  $C_x$  che **assegna un intero ad ogni evento occorso localmente**. Il global clock (nozione

globale del tempo) definito a partire dalle istanze locali dei virtual clock di tutte le entità. Ovviamente i clock virtuali non saranno sincronizzati l'uno con l'altro per i problemi citati nei precedenti paragrafi. Vediamo come, attraverso l'algoritmo, una sincronizzazione dei clock virtuali possibile:

- Prima di computare un evento si aggiorna il clock locale  $C_x = C_x + 1$ ;
- Quando  $x$  invia un messaggio  $M$  include un timestamp  $TS(M) = C_x$
- Quando  $y$  riceve un messaggio da  $M$  da un'altra entità  $y$ , modifica il valore del local counter  $C_x = \max(C_x, TS(M))$  ed esegue il primo passo prima di processare  $M$ .

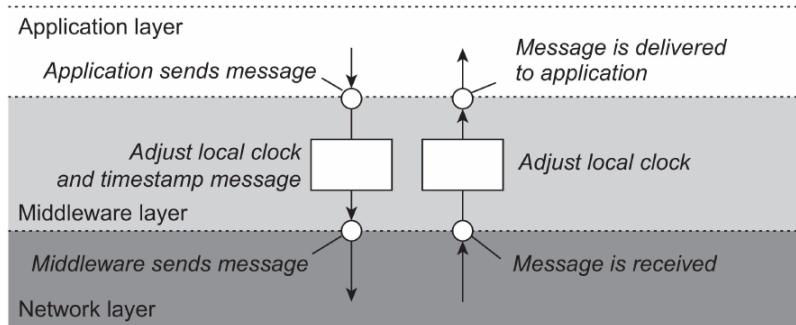


Figure 126: Posizionamento orologi di lamport

### PROBLEMA:

Grazie ai clock logici sappiamo che se  $a \rightarrow b$  allora  $C(a) < C(b)$ .

Se  $C(a) < C(b)$  possiamo dire qualcosa sugli eventi  $a$  e  $b$ ? NO (vedi immagine esempio). Il problema dei clock logici quello che **non catturano relazioni di causalit** e quindi non permettono di inferire informazioni riguardanti le relazioni tra gli eventi, utilizzando i timestamp.

#### 4.9.5 Vector Clocks

I vector clock permettono di avere una rappresentazione più dettagliata di cosa un nodo sa riguardo gli eventi di altre entità (le famose relazioni di causalità che i logical clock non riescono a rappresentare):

- $VC_x[x]$  il numero di eventi che occorrono in  $x$  (clock logico di  $x$ );

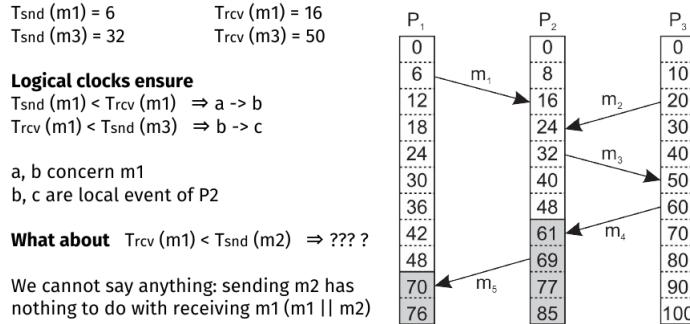


Figure 127: Problema logical clock

- $VC_x[x] = k$  significa che x a conoscenza del fatto che k eventi sono occorsi in y;

Vediamo adesso l'algoritmo:

1. Prima di computare un evento x esegue:  $VC_x[x] = VC_x[x] + 1$  ovvero aggiorna la posizione del vector clock relativa a se stessa ;
  2. Quando x invia un messaggio M, setta il timestamp TS(M) a  $VC_x$  ovvero allega il nuovo timestamp al messaggio.
  3. Quando x riceve un messaggio M da y, modifica il suo vettore locale  $VC_x[i] = \max(VC_x[i], TS(M)[i])$  per ogni k, ed esegue lo step 1 prima di processare il messaggio M.
- Quando y riceve un messaggio da x sappiamo che  $x \Rightarrow TS(M)[x] - 1$  il numero di eventi occorsi ad x prima dell'invio del messaggio.
  - Inoltre x pu indicare a y quanti eventi sono occorsi alle altre entit prima di inviare M (x's view). Infatti TS(M) dice ad y quanti eventi in altre entit hanno preceduto l'invio di M.

FINIRE VECTOR CLOCK PAGINA 45

#### 4.9.6 Mutua esclusione distribuita

Affrontiamo adesso il problema di realizzare l'accesso ad un risorsa condivisa, in **mutua esclusione**, nel contesto di un sistema distribuito. L'idea

ovviamente quella di permettere l'accesso a tale risorsa in modo tale che sia permesso ad una sola entità alla volta.

Un protocollo per la mutua esclusione distribuita è un meccanismo che assicura le seguenti proprietà:

- **Mutua esclusione:** se un entità sta svolgendo operazioni critiche significa l'unica a svolgerle.
- **Fairness:** se un'entità vuole svolgere un'operazione critica, riesce a farlo in un tempo finito.

Esistono due tipologie principali di algoritmi per la mutua esclusione:

- **Token Based:** sono i token che garantiscono l'accesso alle risorse. Esiste un solo token che viene scambiato attraverso la rete. Con questa modalità siamo protetti da deadlock e starvation, tuttavia il token può essere perso.
- **Permission based:** un processo chiede il permesso di accesso ad un altro processo.

I prossimi paragrafi mostrano alcuni di questi algoritmi.

#### 4.9.7 Centralized Algorithm

L'idea molto semplice: stabilito un leader all'interno della topologia, lui che garantisce o meno l'accesso alle risorse. Un'entità semplicemente chiede il permesso al leader ed attende la risposta. Il leader mette in coda altre richieste concorrenti e le serve nell'ordine di arrivo. L'algoritmo corretto ed

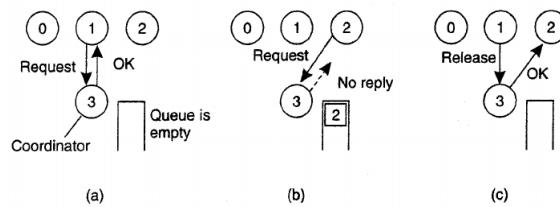


Figure 128: Algoritmo centralizzato

assicura la fairness, inoltre bastano pochi messaggi per accedere alla sezione critica. L'inconveniente maggiore è che il leader rappresenta l'**unico punto di fallimento**.

To access and release a resource we need

- A request from the entity **x** to the leader
- The response of the leader **r** to the request
- A message of the entity to the leader for realising the resource

**n. messages:**  $3d(x, r) < 3 \text{ diam}(G)$

In a complete network: 3 messages

Figure 129: Algoritmo centralizzato: complessit

#### 4.9.8 Traversal Based Algorithm

Questo algoritmo utilizza un token che viaggia in continuazione nella rete e che garantisce l'accesso alla risorsa condivisa. Un'entit **x** semplicemente attende il token, quando ne è in possesso pu accedere alla risorsa, altrimenti inoltra il token ad un'altra entit. L'algoritmo corretto e garantisce la fairness

Algorithms for network traversal, e.g., DF or DF+, DF++

$O(m)$  messages - m number of edges

Using a spanning tree

$O(m)$  to build the spanning tree

$O(n)$  per traversal

Figure 130: Traversal based: complessit

ed è starvation free. Tuttavia il token può sempre essere perso.

Per questo algoritmo esiste anche una versione che assume la topologia di rete essere un anello.

#### 4.9.9 AskAll Algorithm

Anche in questo caso l'idea semplice: un'entit **x** che vuole accedere ad una risorsa chiede il permesso a tutte le altre entit della rete. Se tutti rispondono in senso affermativo, l'entit pu accedere la risorsa. **Requisito** fondamentale per l'algoritmo un **ordinamento totale degli eventi** che pu essere ottenuto

n - 1 $\Rightarrow$ request messages +
n - 1 $\Rightarrow$ OK messages
2 (n - 1) messages are needed to grant 1 access to a resource

Figure 131: Complessit AskAll

utilizzando gli orologi **logici di lamport**. Complessit in figura. L'algoritmo corretto e previene da situazioni di deadlock, starvation ed assicura la fairness. Di contro abbiamo che **se anche uno solo dei nodi fallisce tutte le altre si bloccano**. Inoltre abbiamo bisogno di utilizzare delle primitive che permettano il multicast aumentando la complessità implementativa dell'algoritmo. Un'altra problematica riguarda l'eterogeneità delle macchine connesse ad una rete distribuita: per nodi con capacità computazionale ridotta rispetto agli altri può essere proibitivo l'accesso alla risorsa condivisa.

#### 4.9.10 Quorum-based algorithm

Ancora una volta l'idea di base molto semplice: utilizzare una votazione per guadagnare l'accesso alla risorsa condivisa. le assunzioni di partenza per l'algoritmo sono:

- Ogni risorsa **R** replicata **n** volte;
- ogni replica ha il proprio coordinatore che ne regola l'accesso;
- quando un'entità vuole accedere ad una replica deve avere la maggioranza dei voti  $M \geq n/2$  dei coordinatori.

## 4.10 JBotSim

Poco da dire, guarda le slide e bona.

**m:** the number of coordinators for a resource **R**

An entity **x** needs to send **m** request messages

To have access **x** receives at most **m** response messages

The number of message is at most **2 m**

**Variant:** when an entity **x** is not granted access, it waits for a random time and retries  $\Rightarrow$

at most **2 m k** messages if **x** tries **k** times

Figure 132: Complessit quorum based

## 5 IOT: Protocolli livello applicativo

### 5.1 Introduzione

#### 5.1.1 Mobile Ad-Hoc Network

Una rete di questo tipo un sistema autonomo di host mobili connessi da collegamenti wireless. I nodi sono indipendenti e autonomi (spesso alimentati a batteria) e cooperano tra di loro seguendo il paradigma **peer-to-peer**. Non abbiamo dunque un'infrastruttura di rete fissa e nessun coordinatore centrale, la hanno quindi il vantaggio di adattarsi al contesto applicativo garantendo alta configurabilità.

- Facilmente implementabili;
- facili da configurare;
- Robuste;
- Eterogenee;

Vediamo quindi quali sono le problematiche da tenere in considerazione per questo tipo di rete.

#### Problematiche livello MAC:

- A causa del livello fisico:
  - abbiamo attenuazione del segnale all'aumentare della distanza tra due host;

- non abbiamo dei limiti precisi che indichino la portata delle onde radio.
- abbiamo un rate di errore sui bit trasmessi molto elevato.
- Se consideriamo due entit, queste potrebbero avere canali di comunicazione di differente qualit.
- Il concetto di vicino diverso rispetto alle reti tradizionali: data un'entit, i suoi vicini sono tutte le altre entit nel range di trasmissione.

### Problematiche livello di rete:

- I sistemi di controllo sono distribuiti;
- gestione delle risorse energetiche
- gestione della mobilit:
  - I fallimenti dei link sono pi frequenti;
  - la topologia di rete cambia in modo arbitrario;
  - si riscontrano disconnessioni di nodi e conseguenti partizionamenti della topologia.
- I nodi sono routers: abbiamo bisogno di un protocollo apposito che permetta la comunicazione (multihop routing protocol).
- Gli indirizzi IP non collocano geograficamente i nodi (non sono indicativi della posizione coem avviene per reti fisse), abbiamo bisogno quindi di un meccanismo di routing dinamico.
- i frequenti fallimenti introducono la necessit di un protocollo di aggiornamento delle tabelle di routing. In figura 98 si pu vedere come lo stack di rete cambia in una rete ad-hoc.

#### 5.1.2 WSN: reti di sensori

Prima di entrare nel dettaglio delle WSN vediamo qual' l'approccio tradizionale nell'utilizzo di sensori per rilevazioni ambientali. Lo schema tradizionale vede i sensori, coordinati ad un controller centrale, svolgere un ruolo passivo di semplice rilevazione dell'ambiente.

In una WSN il modello sensibilmente diverso:

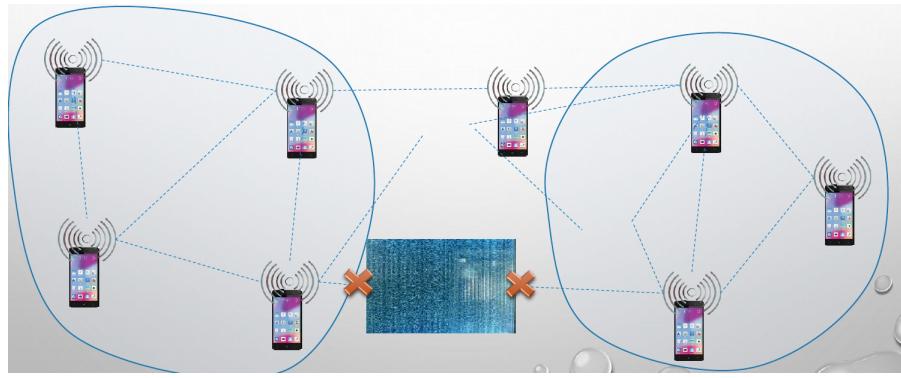


Figure 133: rete ad hoc

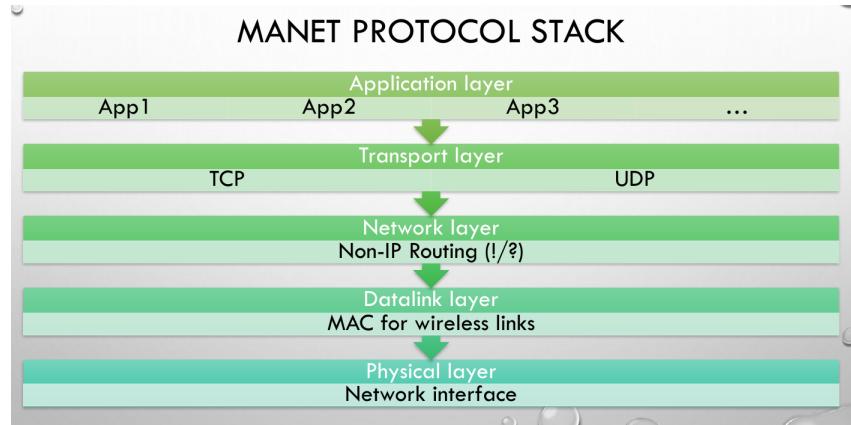


Figure 134: Stack protocolli per rete ad hoc

- i sensori sono **intelligenti**, ovvero sono equipaggiati con microprocessori che permettono anche la computazione dei dati raccolti.
- i sensori sono **collegati tra loro attraverso la tecnologia wireless** (no controller centrale), ci significa che i sensori costituiscono una vera e propria rete distribuita.
- facilmente realizzabili.

Hardware utilizzato per l'implementazione di questo tipo di sensori:

- Processore e memoria;

- Ricevitori radio;
- trasduttori (accelerometro, sensore umidità, luce, gps ecc)
- Convertitore da analogico a digitale
- Batteria (ed eventualmente celle solari);

Ogni sensore **campiona parametri ambientali producendo uno stream di dati**. Solitamente i dati prodotti sono computati dal sensore stesso e inviati ad un nodo che svolge la funzione di **Sink** raccogliendo i dati prodotti dalla WSN. Vediamo i vantaggi per questo tipo di rete:

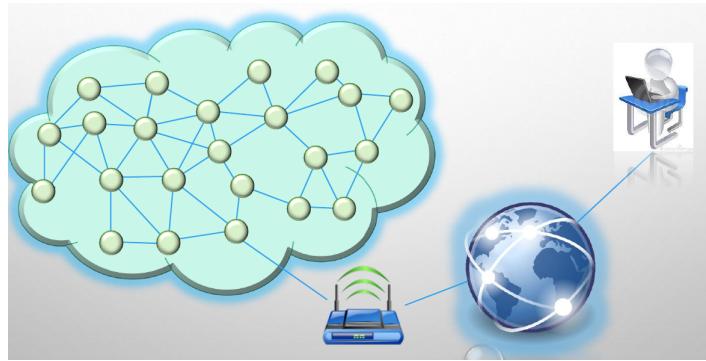


Figure 135: Configurazione tipica per una rete WSN

- L'installazione di questo tipo di reti è molto facile e poco costoso:
  - Non servono cavi;
  - La rete si autoconfigura da sola,
  - Il numero di sensori è scalabile;
  - i sensori possono essere ridondanti (**fault tolerance**)
- alta mobilità dei sensori;
- i sensori possono processare i dati (computazione distribuita di grandi quantità di dati).

Come si differenziano le WSN rispetto alle reti ad-hoc ?

- Il numero di nodi partecipanti la rete generalmente molto più elevato in reti WSN;
- I sensori sono molto limitati rispetto alle risorse energetiche e computazionali,
- Le reti di sensori sono più dense con incidenza di fallimenti molto più elevata rispetto alle reti ad hoc.

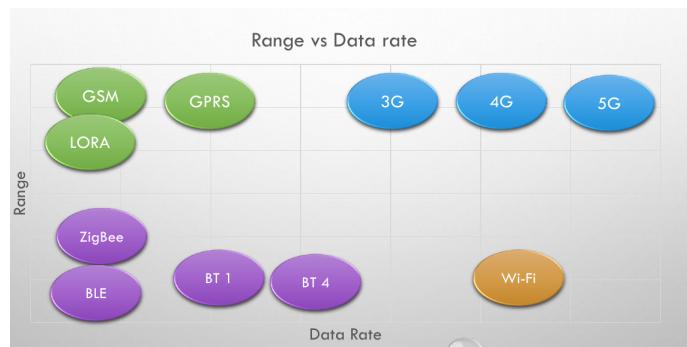


Figure 136: Tecnologie per WSN

Name	ZigBee	WiFi	Bluetooth (1&2)
Standard	802.15.4	802.11 a,b,g	802.15.1
Applications	Monitoring and control	Web, e-mail, video	Cable replacement
System resources	50 to 60 Kbytes	>1 Mbytes	>250 Kbytes
Battery life (days)	100 to > 1000	1 to 5	1 to 7
Network size	64K nodes	~ 100 nodes	7 nodes
Bandwidth (Kbps)	20 to 250 Kbps	~ 100 Mbps	~ 1 Mbps
Maximum transmission range	100+ meters	100 meters	10 meters
Success metrics	Reliability, power, cost	Speed, flexibility	Cost, convenience

Figure 137: Standard a confronto

Vedere esempi RFID.

### 5.1.3 Internet Of The Things

Negli ultimi anni si visto un proliferare di oggetti fisici collegati alla rete ed equipaggiati con apparecchi elettronici (e potenza computazionale). Un esempio indicativo tutta quella branca di dispositivi elettronici indossabili. La maggior parte di questi apparecchi non sono pi nemmeno utilizzati di-

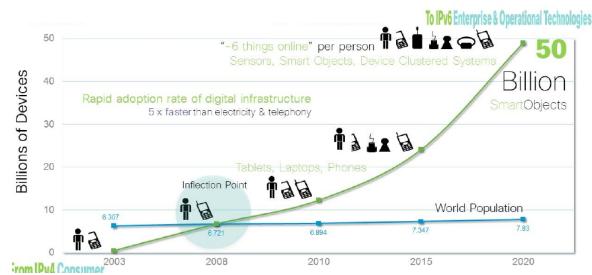


Figure 138: Standard a confronto

rettamente dagli uomini ma hanno una loro logica di lavoro e sono dunque indipendenti da un controllo diretto.

Nell'IOT risulta fondamentale il concetto di **cloud**: i dati sono raccolti dai sensori, inviati al cloud che esegue delle computazioni su questi dati. SI APRE UNA PARENTESI SU MONGODB... MA ANCHE NO.

### 5.1.4 Problemi rilevanti in IOT

SERIE DI CAGATE UNA DIETRO QUELL'ALTRA

## 5.2 MQTT

Gli apparecchi fisici, menzionati nelle sezioni precedenti, ovviamente devono potersi connettere alla rete internet per diventare dispositivi IOT. Ne risulta che per connettersi alla rete devono affidarsi ai protocolli tradizionali (TCP/IP HTTP ecc). Tuttavia lo stack tradizionale pensato per dispositivi che non hanno tutti quei vincoli stringenti riguardo il consumo delle risorse a disposizione.

### 5.2.1 Introduzione MQTT

MQTT (**m**essage **que**uing **t**elemetry **t**ransport) un protocollo che adotta un meccanismo publish/subscribe per l'implementazione di uno scam-

Conventional Internet protocol suite		
Layer	Protocol	Features
Application	HTTP	Application-level access to information, client/server
Transport	TCP/UDP	Communication channels with guarantees (TCP); just an interface to IP (UDP)
Network	IP	Addressing & routing; best-effort

Figure 139: Stack di rete

IoT devices requirements	
Network requirements	Impact on networking
Scalability / redundancy	Multi-hop, mesh networking
Security	Configurable, with different security levels for different devices capabilities
Addressing	Scalability of address space, low overhead on network protocols
Device requirements	Impact on application-level
Low power / battery powered	Low duty-cycle communications
Limited capacity (memory/processor)	Small footprint, low complexity protocols
Low cost	Reliability of the device and further constraints...

Figure 140: Requirements IOT

bio di messaggi affidabile. Il protocollo molto leggero in termini di risorse:

- Utilizzo della banda di rete limitato.
- Overhead dovuto ai pacchetti minimo (meglio di HTTP);

Inoltre:

- implementa architettura client server;
- molto facile da implementare dal lato client, difficile lato server.
- garantisce un servizio affidabile (Quality of Service);
- Data agnostic (un dispositivo che non interessato dalle modalit di ricezione delle informazioni)

- uno standard OASIS;
- **costruito sopra TCP**;

Utilizzato nella comunicazione sensor to satellite, per la home automation  
blablabla.

### 5.2.2 Paradigma Produttore/Consumatore

Questo paradigma di comunicazione ha la caratteristica di essere **loosely coupled** e rappresenta un'alternativa al tradizionale schema di interazione client/server. Gli attori del protocollo sono:

- **Publisher**: produce eventi o condivide dati (interagisce direttamente con il broker);
- **Subscriber**: esprime interesse per un particolare evento (o pattern di eventi), ricevendo notifica dal broker ogniqualvolta quest'ultimo generato (interagisce con il broker);
- Broker (event service): **poich pub e sub sono completamente disaccoppiati in termini di spazio tempo e sincronizzazione**, il broker deve conoscere l'identità di entrambi. Il suo compito di ricevere tutti i messaggi evento da parte del publisher, filtrarli e distribuirli ai subscriber in relazione alle loro richieste. Inoltre deve gestire le richieste di sottoscrizione per tutti gli eventi e ovviamente anche quelle di unsubscription.

Un meccanismo di publish subscribe può essere implementato in diversi modi, solitamente il **broker riveste il ruolo di agente indipendente**. Le operazioni classiche sono:

- Publish;
- Subscribe;
- Notify;
- Unsubscribe;

**Caratteristiche principali:**

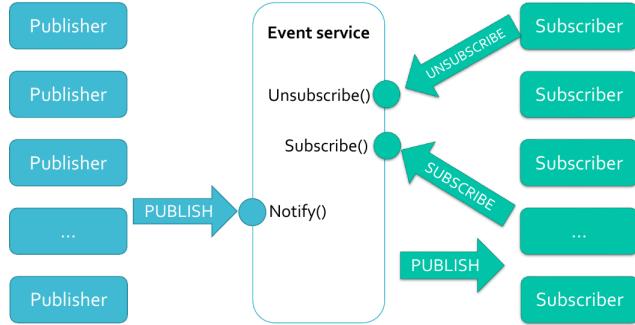


Figure 141: Stack di rete

- **Space decoupling:** pub e sub non hanno bisogno l'uno dell'altro (non conoscono ip o porta l'uno dell'altro);
- **Time decoupling:** pub e sub non hanno bisogno di eseguire nello stesso momento per interagire.
- **Synchronization decoupling:** le operazioni di pub e sub non sono interrotte durante la pubblicazione o la ricezione delle informazioni.
- **Scalabilit:** le operazioni sul broker possono essere parallelizzate e sono event-driven (verranno eseguite solo se necessario al presentarsi di un evento). In questo senso il paradigma molto migliore rispetto all'approccio client server.

Come detto i messaggi devono essere filtrati dal broker sulla base di:

- **Topic:** il soggetto del topic contenuto come parte del messaggio (semplice stringa).
- **Contenuto:** il client si iscrive ad una specifica query (temperatura  $\geq 30$ ). Se vogliamo questo tipo di sottoscrizione i dati non possono ovviamente essere crittati.
- **Tipo:** filtraggio basato sia sul contenuto che sulla struttura.

Nonostante il paradigma permetta a pub e sub di essere loosely coupled, come già specificato, importante sottolineare che comunque **quest'ultimi devono trovarsi d'accordo sul topic prima di ogni altra cosa**. Inoltre il publisher non può sapere se l'evento viene consumato da un subscriber (spreco di risorse).

### 5.2.3 Produttore/consumatore in MQTT

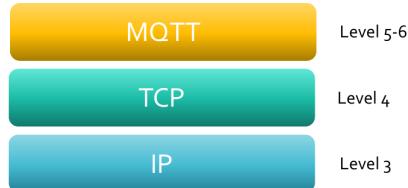


Figure 142: Stack con MQTT

Prima di vedere come avvengono le connessioni in MQTT utile introdurre i campi presenti nell'**header di un messaggio CONNECT di questo protocollo**.

- **Client ID:** una stringa che identifica univocamente il client. Il campo pu essere lasciato vuoto in questo caso il broker ne assegna uno ma la sessione non sar persistente (il campo Clean session deve essere a 1);
- **Clean session:** Indica la volont del client di avere o meno una sessione persistente (false) (significa che il broker salva le sottoscrizioni e i messaggi per il client). Se esiste una sessione precedente viene ripristinata altrimenti, se il bit a true, vengono eliminate.
- **Username/Password:** header non criptati (a meno di protocolli appositi a livello trasporto)
- **Will flags:** se un client si disconnette in modo improvviso, il broker invia questi messaggi per avvisare gli altri client.
- **KeepAlive:** pacchetti utilizzati per comunicare al broker che il client attivo (inviai dal client).

I messaggi **CONNECT** sono utilizzati dai client per richiedere la connessione con il broker, alla ricezione, quest'ultimo:

1. Il broker invia un **ack** utilizzando un messaggio **CONNECTACK**. Il messaggio contiene parametri che confermano o meno la riuscita della connessione al client. Il campo **session present** indica se il broker aveva una sessione salvata relativa al client.

2. Dopo la connessione un client pu pubblicare messaggi. I messaggi contengono un **topic** (utilizzato per filtrare i messaggi per i subscribers ) e il **payload** (solitamente in formato **JSON**).

I messaggi **PUBLISH** hanno nell'header i seguenti parametri:

- **Packet id:** id del pacchetto.
- **topicName:** stringa che identifica il topic da pubblicare.
- **qos:** 0,1,2;
- **retainFlag:** informa il broker se il messaggio deve essere salvato come ultimo valore noto per il topic relativo. Se un client sottoscrive il topic solo adesso ricever questo messaggio. I messaggi di retain non hanno nulla a che vedere con le sessioni persistenti sono salvati dal broker indipendentemente dal fatto che questi siano consegnati o meno. Rappresentano lo stato dell'arte per un particolare topic. Questo tipo di messaggi hanno particolarmente senso se si prevede che il topic non sarà aggiornato molto frequentemente
- **Payload.**
- **dupFlag:** indica se il messaggio è un duplciato di uno precedente che non ha ricevuto ack da parte del broker ( ha importanza solo se la qos > 0)

Quando un broker riceve un messaggio **PUBLISH**:

1. invia un ack se richiesto dalla qos;
2. processa il messaggio.
3. invia il messaggio ai propri subscribers.

Per la struttura dei messaggi di **SUBSCRIBE** e **SUBACK** vedere le slide. I **TOPIC** sono stringhe organizzate in modo gerarchico (home/firstfloor/bedroom/presence). Si possono usare wildcards come + e cancelletto. I path preceduti da dollaro sono utilizzati per le statistiche interne di MQTT possono letti ma non pubblicati dai client (\$SYS/broker/clients/connected)). **non esistono regole specifiche per la pubblicazione dei topic**, tuttavia esistono delle "buone pratiche" da seguire:

- non iniziare un topic con /
- non usare spazi
- tenere i topic abbastanza corti
- Utilizzare sono caratteri ASCII UTF-8
- topic specifici sono preferiti rispetto quelli generici
- non pubblicare topic con #. Il workload per il client potrebbe essere troppo alto.

#### 5.2.4 QOS in MQTT

La qualit del servizio un accordo tra il mittente e il destinatario del messaggio: nel nostro caso l'**accordo** tra **publisher** e **subscriber** (utilizzata anche tra pub e broker e tra sub e broker). I livelli per la qos sono:

- **at most once (0)**: best effort (no ack, messaggi non salvati nel broker ma inviati direttamente). Garantisce lo stesso livello di qos di TCP, tuttavia TCP garantisce la consegna solo in presenza di una connessione, se uno dei peers si disconnette non abbiamo pi questa garanzia.
- **at least once(1)**: i messaggi sono numerati e salvati nel broker prima di essere inviati. Si utilizzano ack (PUBACK) per confermare o meno la ricezione. Ovviamente **un messaggio pu essere consegnato pi di una volta**
- **exactly once (2)**: essendo il livello pi alto introduce l'overhead maggiore nella comunicazione. Si utilizza un **doppio handshake a 2 vie** per garantire che il messaggio sia consegnato una e una sola volta al destinatario (vedere slide. Comunque da figura si capisce perch necessario). Qos 1 e 2 richiedono sempre l'utilizzo di una sessione persistente. La scelta della qos deve essere fatta consequentemente alle necessit applicative, con un occhio di riguardo al carico di lavoro che i client devono svolgere (pi qos implica pi overhead).

Un **flag particolare**, presenti nei pacchetti CONNECT quello di **last will**. Viene specificato dal client al momento delle connessione con il broker cos da comunicargli le sue ultime volont in caso di disconnessione improvvisa



Figure 143: MQTT: qos 0

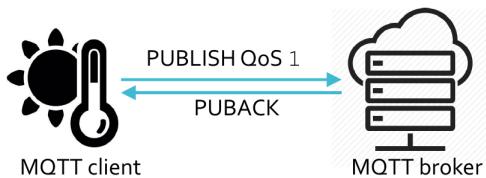


Figure 144: MQTT: qos 1

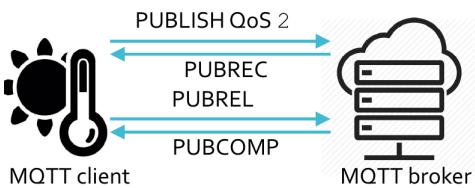


Figure 145: MQTT qos 2

(solitamente un topic da pubblicare). Il broker salva il messaggio e lo recapita a tutti i subscribers alla disconnessione del publisher. Al contrario, se il client si disconnette questo messaggio è eliminato. Per un messaggio di last will si possono impostare gli stessi flag di un topic normale (lastwill-topic, lastwillqos, lastwillmessage, lastwillretain). Anche il **broker può utilizzare messaggi last will** quando:

- si verifica un errore I/O di rete
- un client non invia messaggi keepalive in tempo;
- il broker chiude la comunicazione con un client a causa di un errore nel protocollo.

I messaggi lastwill sono spesso utilizzati insieme ai messaggi di retain (se retain ON, last will può essere utilizzato per pubblicare un ratain con payload

OFF al momento della disconnessione). Messaggi di keepalive in figura (per dettagli slide). Stessa cosa per struttura dei pacchetti. Alcuni **problemi**

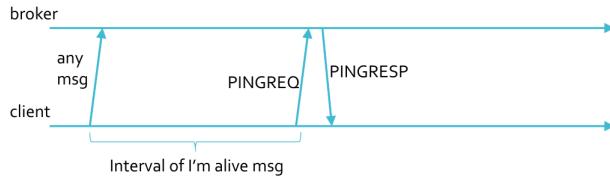


Figure 146: MQTT:keepalive

Structure of an MQTT control packet:

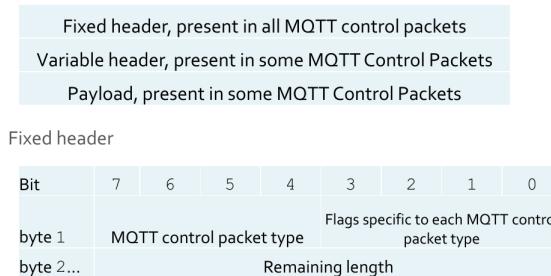


Figure 147: MQTT:struttura pacchetti

che riguardano MQTT:

- Il bisogno di un broker centralizzato pu essere limitante in applicazioni IOT distribuite: l'overhead sul broker potrebbe risultare incompatibile con le sue risorse computazionali, qualora la rete considerevolmente le proprie dimensioni.
- Il broker rappresenta un **punto unico di fallimento**, poich necessario al mantenere il sistema attivo.
- MQTT si affida, al livello sottostante, a TCP che non risulta essere particolarmente leggero se rapportato alle capaci dei device IOT. Infatti richiede molto pi risorse rispetto a UDP, i tempi per stabilire una connessione sono pi lunghi. Questa richiesta di risorse consuma molta energia, ci pu essere inaccettabile se ci si rapporta a dispositivi con batteria.

## 5.3 COAP

### 5.3.1 Introduction

Nell'internet of the things la rete non è semplice veicolo di messaggio ma utilizzata per realizzare e comporre servizi.

### 5.3.2 Coap

Coap è un protocollo a livello applicazione per un constrained environment. Per constrained intendiamo la quantità ridotta di risorse che caratterizza i dispositivi IOT. L'idea dunque quella di permettere comunicazione cooperazione di questo tipo di dispositivi con modalità e costrutti tipici di HTTP, che per rappresenterebbe una soluzione troppo pesante nel contesto citato. A differenza di HTTP il protocollo **completamente asincrono**. Coap è un

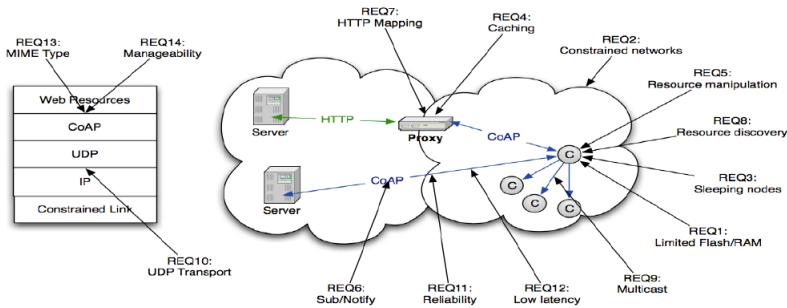


Figure 148: Pila COAP

efficiente protocollo REST ideale per dispositivi e reti limitate e specializzato nella comunicazione M2M.

Coap può essere utilizzato per mediare una comunicazione tra la rete web, che utilizza la pila classica di rete, ad un ambiente constrained. L'idea quella di utilizzare un nodo sink che implementi COAP e che comunichi con l'esterno della rete, in questo modo i nodi, comunicano tra di loro tramite coap, ma hanno comunque un'interfaccia per comunicare con l'esterno. COAP è realizzato per comunicare attraverso una logica REST ed è per questo che comunica bene con HTTP, di cui tuttavia non rappresenta un sostituto. **COAP mette quindi a disposizione un protocollo di trasferimento embedded (coap: invece che http:)**

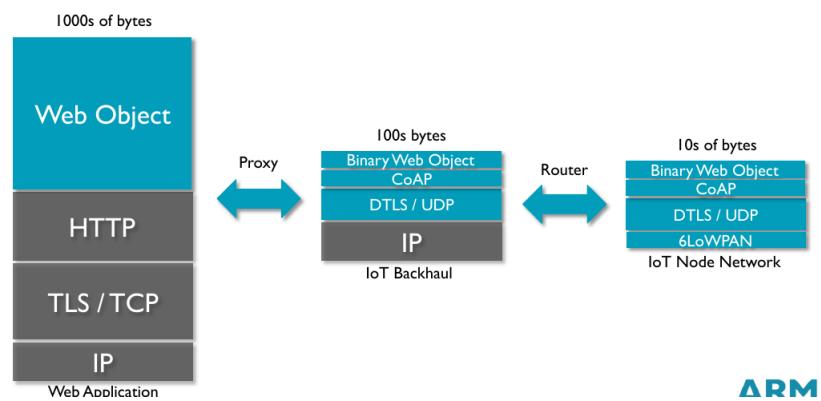


Figure 149: Dal web ad un constrained environment