

Appunti di Distributed Algorithms

Emanuele Gentiletti

Indice

1	Introduzione	5
2	Modello di rete sincrona	7
2.1	Sistemi di rete sincroni	7
2.2	Fallimento	9
	Stopping failure	9
	Fallimento bizantino	9
	Fallimento di collegamento	9
2.3	Input e output	9
2.4	Esecuzioni	10
2.5	Metodi di dimostrazione	10
	Asserzione di invarianti	10
	Simulazioni	10
2.6	Misure di complessità	11
	Complessità temporale	11
	Complessità di comunicazione	11
2.7	Randomizzazione	11
3	Elezione del leader in un anello sincrono	13
3.1	Il problema	13
3.2	Impossibilità per processi identici	14
3.3	LCR - Un algoritmo di base	14
	Algoritmo LCR (pseudocodice)	15
	Algoritmo LCR (formale)	15

Capitolo 1

Introduzione

Il termine *algoritmi distribuiti* include una grande varietà di algoritmi concorrenti usati per una vasta gamma di applicazioni. Originalmente, il termine si riferiva ad algoritmi progettati per essere eseguiti su molti processori “distribuiti” su una vasta area geografica. Nel corso degli anni, date le similarità riscontrate tra queste categorie, il termine racchiude anche gli algoritmi eseguiti su LAN e su multiprocessori a memoria condivisa.

Il termine “processori” suggerisce che si stia trattando di parti hardware. Ai fini di questo trattato è più utile pensare a questi come processi software in esecuzione sui processori hardware.

Ci sono molti tipi di algoritmi distribuiti. Alcuni degli attributi per cui si differenziano sono:

- *Il modello di comunicazione tra processi (IPC)*: Gli algoritmi distribuiti sono eseguiti su più processori, che hanno bisogno di comunicare in qualche modo.
- *Il modello di temporizzazione*: I processori possono essere considerati:
 - **completamente sincroni**: eseguono comunicazioni e computazioni di pari passo.
 - **completamente asincroni**, eseguono passi a velocità arbitrarie e in ordine arbitrario.
 - **parzialmente sincroni**: i processori hanno informazioni parziali sulla temporizzazione degli eventi (es. i processori hanno dei limiti sulle loro velocità relative o hanno accesso a un clock approssimativamente sincronizzato).
- *Il modello di fallimento*: Si può supporre che il sistema sia completamente affidabile, o che l'algoritmo debba tollerare una certa quantità di comporta-

menti errati. I processi possono interrompersi, senza avvertimento, oppure possono esibire un più grave *fallimento bizantino*, dove si comportano in maniera arbitraria.

- I *problemi affrontati*: Ad esempio allocazione di risorse, comunicazione, consenso tra processori distribuiti, controllo della concorrenza in un database, rilevamento dei deadlock, snapshot globali, e implementazione di diversi tipi di oggetto.

Il comportamento degli algoritmi distribuiti è spesso difficile da comprendere per molti fattori dovuti da un maggiore tasso di *incertezza e indipendenza delle attività*. Alcuni dei tipi di incertezza con cui gli algoritmi devono confrontarsi sono:

- numero di processori non noto
- topologia di rete non nota
- input indipendenti in diverse locazioni
- diversi programmi in esecuzione tutti insieme, avviati in diversi momenti, e in operazione a diverse velocità
- non determinismo del processore
- tempi di arrivo dei messaggi incerti
- fallimenti dei processori e delle comunicazioni

Negli algoritmi distribuiti, invece di comprendere tutto sul loro comportamento, il meglio che si può fare spesso è capire alcune proprietà specifiche e certe del loro comportamento.

Capitolo 2

Modello di rete sincrona

2.1 Sistemi di rete sincroni

Un *sistema di rete sincrono* consiste di una collezione di processi locati ai nodi di un grafo orientato di rete.

Per definire un sistema di rete sincrono formalmente, si inizia con un grafo diretto $G = (V, E)$. Si usa la lettera n per indicare $|V|$, il numero di nodi nel grafo di rete.

Per ogni nodo i di G , si usano le notazioni:

- $out-nbrs_i$ per indicare i *vicini in uscita* di i , ovvero quei nodi che hanno archi entranti che partono da i
- $in-nbrs_i$ per indicare i *vicini in entrata* di i , ovvero quei nodi da cui partono archi entranti in i .
- $distance(i, j)$ indica la lunghezza del cammino minimo tra i e j in G , se esiste. Altrimenti, $distance(i, j) = \infty$
- $diam$, il *diametro*, indica la massima $distance(i, j)$ tra tutte le coppie di nodi (i, j)

Si suppone anche di avere un alfabeto fisso di messaggi M , e si considera *null* un segnaposto che indica l'assenza di un messaggio.

Associato a ogni nodo $i \in V$, si ha un *processo*, che consiste formalmente dei seguenti componenti:

- $states_i$, un (non necessariamente finito) insieme di *stati*
- $start_i$, un sottoinsieme non vuoto di $states_i$, indicato come gli *stati iniziali*
- $msgs_i$, una *funzione di generazione dei messaggi*, che associa $states_i \times out-nbrs_i$ a elementi di $M \cup \{null\}$.
- $trans_i$, una *funzione di transizione di stato* che associa a $states_i$ e a vettori di elementi $M \cup \{null\}$ (indirizzati da $in-nbrs_i$) a $states_i$

Ogni processo ha quindi un insieme di stati, di cui un sottoinsieme è di stati iniziali. L'insieme di stati non deve essere finito, permettendo quindi di modellare sistemi che includono strutture dati illimitate come contatori.

Funzione di generazione dei messaggi specifica, per ogni stato e vicino in uscita, il messaggio (o *null*) che il processo i invia al vicino indicato, a partire dallo stato dato.

Funzione di transizione di stato specifica, per ogni stato e collezione di messaggi da tutti i vicini in entrata, il nuovo stato verso cui i si sposta.

Canale (o link) una locazione associata a ogni arco (i, j) che può, in ogni momento, contenere al massimo un singolo messaggio di M .

L'esecuzione dell'intero sistema inizia con tutti i processi in stati iniziali arbitrari, e con tutti i canali vuoti. Poi i processi, insieme e passo per passo, eseguono i seguenti due passi:

1. Applicano la funzione di generazione dei messaggi allo stato corrente per generare i messaggi da inviare ai vicini in uscita, e mettono questi messaggi nei canali appropriati.
2. Applicano la funzione di transizione di stato allo stato corrente e ai messaggi in entrata per ottenere il nuovo stato, rimuovendo i messaggi dai canali.

La combinazione dei due passi è chiamata *round*. Notare che, in generale, non vengono imposte restrizioni sull'ammontare di computazione che un processo deve eseguire per computare i valori delle sue funzioni di generazione dei messaggi e di transizione di stato.

Notare anche che il modello presentato è deterministico, nel senso che le funzioni di generazione dei messaggi e di transizione di stato hanno valori singoli. Per cui, a partire da una particolare combinazione di stati iniziali, la computazione si svolge in modo univoco.

Stati di halt L'halt dei process può essere tenuto in considerazione in questo modello designando alcuni stati dei processi come *stati di halt*, e specificando che da questi stati non ci può più essere alcuna attività, ovvero: nessun messaggio viene più generato, e l'unica transizione possibile è un loop verso lo stato stesso. Questi stati non svolgono lo stesso ruolo che hanno negli automi finiti, e *non* sono quindi considerati *di accettazione*, ma hanno il solo scopo di interrompere il processo. Ciò che viene computato dal processo deve essere determinato in un altro modo.

Nodo e processo di ambiente Occasionalmente, si vuole considerare un sistema sincrono in cui i processi si avviano in diversi round. Si modella questa situazione estendendo il grafo di rete per includere uno speciale *nodo di ambiente*, che ha archi verso tutti i nodi ordinari. Lo scopo del corrispondente *processo di ambiente* è di inviare degli speciali *messaggi di wakeup* a tutti gli altri processi. Ogni stato iniziale di ognuno degli altri processi deve essere *quiescente*, che significa che non causa la generazione

di messaggi, e che può essere cambiato a uno stato diverso solo in seguito alla ricezione di un messaggio di wakeup dall'ambiente o da un messaggio non nullo da un altro processo. Per cui, un processo può essere svegliato:

- **direttamente**, da un messaggio di wakeup dall'ambiente
- **indirettamente**, da un messaggio non nullo da un altro processo.

Grafi non orientati A volte si vuole considerare il caso in cui il grafo di rete non è diretto. Questa situazione viene modellata considerando un grafo diretto con archi bidirezionali tra tutte le coppie di vicini. In questo caso, si usa la notazione $nbrs_i$ per indicare i vicini di i nel grafo.

2.2 Fallimento

Si considerano diversi tipi di fallimento nei sistemi sincroni, tra cui *fallimento dei processi* e *fallimento dei collegamenti*.

Stopping failure

Il processo si interrompe nel mezzo della sua esecuzione. Nei termini del modello, il processo potrebbe fallire prima o dopo aver eseguito una qualche istanza del Passo 1 o Passo 2 descritti prima. In aggiunta, potrebbe interrompersi nel mezzo del Passo 1. Questo significa che il processo potrebbe riuscire a mettere nei canali solo un sottoinsieme dei messaggi che dovrebbe produrre. Si dà per scontato che questo possa essere un *qualunque sottoinsieme* - non si considera che il processo produce messaggi sequenzialmente e che fallisce nel mezzo della sequenza.

Fallimento bizantino

Il processo può generare i suoi prossimi messaggi e il suo prossimo stato in maniera arbitraria, senza necessariamente seguire le regole stabilite dalle funzioni di generazione dei messaggi e di transizione di stato.

Fallimento di collegamento

Il collegamento può fallire perdendo messaggi. Nei termini del modello, un processo può tentare di inserire un messaggio nel canale durante il Passo 1, mentre il collegamento fallace non lo registra.

2.3 Input e output

Si usa la convenzione di codificare gli input e gli output negli stati. In particolare, gli input sono inseriti in variabili designate negli stati iniziali. Il fatto che il processo possa avere diversi stati iniziali è importante qui, perché permette di accomodare diversi input. Di fatto, si dà normalmente per scontato che l'unica

fonte di molteplicità di stati iniziali è la possibilità di avere diversi valori di input nelle variabili di input.

2.4 Esecuzioni

Per ragionare sul comportamento di un sistema di rete sincrono, è necessaria una notazione formale dell'esecuzione di un sistema.

- Un *assegnazione di stato* C_r di un sistema è definita come l'assegnazione di uno stato a ogni processo del sistema al round r .
- Un *assegnazione di messaggio* è un'assegnazione di messaggi (possibilmente *null*) a ogni canale del sistema. M_r e N_r rappresentano rispettivamente i messaggi inviati e ricevuti al round r nel sistema, e possono differire nel caso in cui ci sia perdita di messaggi.

Un'esecuzione del sistema è definita come una sequenza infinita

$$C_0, M_1, N_1, C_1, M_2, N_2, C_2, \dots$$

Ci si riferisce a C_r anche come l'assegnazione di stato che avviene al *tempo* r , ovvero il punto dopo cui r round si sono susseguiti.

Se α e α' sono due esecuzioni del sistema, si dice che α è *indistinguibile* da α' rispetto al processo i (denotato $\alpha \stackrel{i}{\sim} \alpha'$) se i ha la stessa sequenza di stati, la stessa sequenza di messaggi in uscita e la stessa sequenza di messaggi in entrata sia in α che in α' . Se le esecuzioni condividono queste caratteristiche solo fino al round r , si può dire che le esecuzioni sono *indistinguibili rispetto a i nel corso di r round*. Queste definizioni vengono estese anche alla situazione in cui le esecuzioni da comparare sono esecuzioni di due differenti sistemi sincroni.

2.5 Metodi di dimostrazione

Asserzione di invarianti

Un'asserzione di invariante è una proprietà dello stato del sistema (in particolare, dello stato di tutti i processi) che resta vera in ogni esecuzione, dopo ogni round. Si può definire un'asserzione che sia vera dopo un certo numero di round. Le asserzioni di invarianti per sistemi sincroni sono generalmente dimostrate per induzione su r , il numero di round completati, a partire da $r = 0$.

Simulazioni

L'obiettivo è mostrare che un algoritmo sincrono A “implementa” un altro algoritmo sincrono B , nel senso che hanno lo stesso comportamento in termini di input/output. La corrispondenza tra A e B è espressa tramite un'asserzione che mette in relazione gli stati di A e B , quando i due algoritmi vengono eseguiti

con gli stessi input e vengono eseguiti con le stesse modalità di fallimento per lo stesso numero di round. Un'asserzione di questo tipo è conosciuta come una *relazione di simulazione*. Come per le invarianti di asserzione, le relazioni di simulazione sono generalmente dimostrate per induzione sul numero di round completati.

2.6 Misure di complessità

Complessità temporale

Viene misurata nei termini del numero di round richiesti fino a quando tutti i round richiesti vengono prodotti, o finché tutti i processi si arrestano. Se il sistema permette tempi di avvio variabili, la complessità viene misurata dal primo round in cui avviene un *wakeup*, in qualunque processo esso occorra.

Nella pratica, la complessità temporale è più importante, non solo per gli algoritmi sincroni ma per tutti gli algoritmi distribuiti.

Complessità di comunicazione

Viene misurata nei termini della quantità di messaggi non-nulli che viene inviata. Occasionalmente, si tiene anche in conto del numero dei bit nei messaggi.

La complessità di comunicazione è significativa se causa abbastanza congestione da rallentare l'elaborazione. L'impatto del carico della comunicazione sulla complessità temporale può accumularsi nel caso frequente in cui vi sono più algoritmi distribuiti in esecuzione in una stessa rete, condividendo la stessa banda. È difficile quantificare l'impatto che i messaggi di un algoritmo hanno sulla performance di altri algoritmi, per cui ci si limita ad analizzare (e cercare di minimizzare) il numero di messaggi generati dagli algoritmi individuali.

2.7 Randomizzazione

Invece di richiedere che i processi siano deterministici, a volte è utile permettere a questi di eseguire scelte casuali, basate su qualche distribuzione probabilistica. Dal momento in cui il modello sincrono di base non lo permette, aumentiamo il modello introducendo una nuova *funzione random* in aggiunta a quelle di generazione dei messaggi e di transizione, per rappresentare il passo della scelta casuale.

Formalmente, si aggiunge una componente $rand_i$ a ogni nodo i nella descrizione dell'automa fornita in precedenza. Per ogni stato s , $rand_i(s)$ è una distribuzione probabilistica su di un sottoinsieme di $states_i$.

In ogni round di esecuzione, la funzione $rand_i$ viene usata inizialmente per scegliere nuovi stati, e poi le funzioni $msgs_i$ e $trans_i$ vengono applicate come al solito.

L'esecuzione del sistema è definita come una sequenza infinita

$$C_0, D_1, M_1, N_1, C_1, D_2, M_2, N_2, C_2, \dots$$

dove ogni C_r e D_r sono *assegnazioni di stato* e dove ogni M_r e N_r sono *assegnazioni di messaggio*. D_r rappresenta i nuovi stati dei processi dopo le scelte casuali del round r .

Le affermazioni su cosa viene computato da un sistema randomizzato sono solitamente probabilistici, e asseriscono che certi risultati vengono raggiunti con almeno un certo grado di probabilità. Quando queste affermazioni vengono fatte, si intende generalmente che queste valgano per tutti gli input, e, in caso di sistemi soggetti a fallimento, per tutti i pattern di fallimento.

Per modellare gli input e i pattern di fallimento, si dà per presupposto che un'entità fittizia chiamata *avversario* controlli le scelte di input e le occorrenze dei fallimenti, e le affermazioni probabilistiche asseriscono che il sistema si comporta in modo corretto in competizione con ogni avversario ammissibile.

Capitolo 3

Elezione del leader in un anello sincrono

3.1 Il problema

Si suppone che il grafo di rete G sia un anello che consiste di n nodi, numerati da 1 a n in senso orario. Si conta spesso $\text{mod } n$, permettendo a 0 di essere un altro nome per il processo n , $n + 1$ per il processo 1, e così via.

Presupposti:

- I processi associati con i nodi di G non conoscono i propri indici, o quelli dei propri vicini.
- Le funzioni di generazione dei messaggi e di transizioni sono definite nei termini di nomi locali e relativi per i vicini.
- Ogni processo è in grado di distinguere il proprio vicino in senso orario da quello in senso antiorario.

Il requisito è che, eventualmente, esattamente un processo dovrebbe dare in output la decisione per cui esso è il leader, ad esempio cambiando una componente del suo stato chiamata *status* al valore *leader*.

Ci sono diverse versioni del problema:

1. Si richiede che tutti i processi non leader diano eventualmente in output il fatto che non sono leader, ad esempio cambiando il proprio componente *status* al valore *non-leader*.
2. L'anello può essere unidirezionale o bidirezionale. Se unidirezionale, allora ogni arco è diretto da un processo al suo vicino in senso orario, per cui i messaggi possono essere inviati solo in senso orario.
3. Il numero n di nodi nell'anello può essere conosciuto o sconosciuto.

- Se è conosciuto, significa che i processi devono comportarsi correttamente solo in anelli di dimensione n , e quindi che possono usare il valore n nei loro programmi.
 - Se è sconosciuto, significa che i processi devono essere in grado di lavorare in anelli di diverse dimensioni, per cui non possono usare informazioni sulla dimensione dell'anello.
4. I processi possono essere identici o distinguersi iniziando ognuno con un *identificatore univoco (UID)* scelto da uno spazio grande e completamente ordinato di identificatori, come \mathbb{N}^+ . Si dà per vero che l'UID di ogni processo sia diverso da ogni altro nell'anello, ma che non ci siano restrizioni su quali UID possano apparire nell'anello. Gli identificatori possono essere ristretti per essere manipolati solo tramite certe operazioni, come comparazioni, o possono ammettere operazioni senza restrizioni.

3.2 Impossibilità per processi identici

Una prima e semplice osservazione è che se tutti i processi sono identici, il problema non può essere risolto dato il modello proposto. Questo anche nel caso in cui l'anello sia bidirezionale e la sua dimensione è conosciuta dai processi.

Teorema 3.1. *Sia A un sistema di processi, $n > 1$, ordinati in un anello bidirezionale. Se tutti i processi in A sono identici, allora A non risolve il problema dell'elezione del leader.*

Dimostrazione. Si supponga che esista il sistema A descritto che risolve il problema dell'elezione del leader. Si ottiene una contraddizione.

Si può dire senza nessuna perdita di generalità che ognuno dei processi di A ha esattamente uno stato iniziale. Questo perché se ogni processo avesse più di uno stato iniziale, si potrebbe scegliere uno qualunque degli stati iniziali e ottenere una nuova soluzione in cui ogni processo ha un solo stato iniziale. Con questo presupposto, A ha esattamente una esecuzione.

Si consideri quindi l'univoca esecuzione di A . È semplice verificare, per induzione sul numero di round r che sono stati eseguiti, che tutti i processi sono in stati identici immediatamente dopo r round. Per cui, se uno dei processi dovesse raggiungere uno stato in cui *status* corrisponde a *leader*, allora tutti gli altri processi raggiungerebbero lo stesso stato allo stesso momento, e questo violerebbe il requisito di unicità.

□

3.3 LCR - Un algoritmo di base

- Usa solo comunicazione unidirezionale
- Non necessita la conoscenza della dimensione dell'anello

- L'unico output viene dal leader.
- Vengono usate solo operazioni di comparazione sugli UID.

Algoritmo LCR (pseudocodice)

```

process A(int my_id) {

    <invia my_id al prossimo nodo nell'anello>

    while(true) {

        received_id = <ricevi dal nodo precedente>

        if(received_id > my_id) {
            <invia received_id al nodo successivo>
        }
        else if(received_id == my_id) {
            output = leader
            break
        }

    }
}

```

In questo algoritmo, il processo con l'UID più grande è l'unico a dare come output leader.

Algoritmo LCR (formale)

L'alfabeto dei messaggi M è l'esatto insieme degli UID.

Per ogni i , lo stato in $states_i$ corrisponde alle seguenti componenti:

- u , uno UID, inizialmente l'UID di i
- $send$, uno UID o $null$, inizialmente lo UID di i
- $status$, con possibili valori $\{unknown, leader\}$, inizialmente $unknown$

L'insieme dei messaggi iniziali $start_i$ consiste nel singolo stato definito dalle assegnazioni date.

Per ogni i , la funzione di generazione dei messaggi $msgs_i$ è definita come segue:

invia il valore corrente di $send$ al processo $i + 1$

In realtà, i nodi non conoscono il valore $i + 1$, se non con un nome relativo come *vicino in senso orario*. Viene riportato qui in questo modo per semplicità.

Per ogni i , la funzione di transizione è definita come segue:

```

send := null
if (il messaggio in arrivo è v, uno UID)
  case
    v > u : send := v
    v = u : status := leader
    v < u : noop
  endcase

```

La prima riga pulisce lo stato dagli effetti dell'invio del precedente messaggio.

La descrizione dell'algoritmo ha una traduzione diretta in una macchina a stati per processi descritta nel Capitolo 2:

- Ogni stato consiste nel valore di ognuna delle variabili.
- Le transizioni possono essere descritte in termini di cambiamenti alle variabili.

Per dimostrare che l'algoritmo è corretto, bisogna dimostrare che esattamente un processo esegua eventualmente *leader* come output.