Garanzia di Consistenza: Implementazione dell'algoritmo di Multicast totalmente e causalmente Ordinato in un Datastore Key-Value

Lorenzini Marco
Magistrale Ingegneria Informatica
Anno Accademico 2023-2024
Roma, Italia
marcolorenzini23@gmail.com

Abstract

Nei sistemi distribuiti, garantire la coerenza tra i nodi del sistema attraverso l'utilizzo di meccanismi di consistenza è di fondamentale importanza. Questo studio analizza due forme di consistenza: sequenziale e causale, implementate rispettivamente tramite l'algoritmo di Multicast Totalmente e Causalmente Ordinato. Esaminiamo l'efficacia di tali approcci nell'assicurare un flusso coerente di dati all'interno del database Key-Value, valutando la gestione della sequenza temporale degli eventi e delle dipendenze causali.

Parole Chiave: *consistenza causale, consistenza sequenziale, multicast totalmente ordinato, multicast causalmente ordinato, clock vettoriali, clock scalari, sincronizzazione.*

1. Introduzione

Ordinare gli eventi che occorrono nei sistemi distribuiti moderni è vitale per garantire l'affidabilità e la coerenza dei dati. Nei sistemi distribuiti asincroni, non è sempre possibile mantenere limitata l'approssimazione dei clock fisici, per questo motivo il timestamping non può basarsi sul tempo fisico, ma viene basato sul tempo logico. Questa pratica è fondamentale per garantire la coerenza delle operazioni. Gli algoritmi di Multicast Causalmente Ordinato e di Multicast Totalmente Ordinato, oggetto di questa ricerca, si basano su questa fondamentale nozione. Analizzeremo approfonditamente il modo in cui tali algoritmi gestiscono la consistenza e la loro implementazione.

1.1 Clock logici

La consistenza sequenziale è consentità grazie all'utilizzo dei clock logici scalari, mentre la consistenza causale è consentità grazie all'uso dei clock logici vettoriali. L'idea alla base dei clock logici è che:

1. Due eventi che occorrono sullo stesso processo p_i si sono verificati esattamente nell'ordine in cui p_i li ha osservati. Indichiamo la relazione di ordinamento tra due eventi che occorrono sullo stesso processo i come $\rightarrow_i$.
2. Quando un messaggio viene inviato da p_i a p_j, l'evento di send precede l'evento di reiceive. Nel 1978 Lamport introduce il concetto di relazione happened-before che stabilisce che 2 eventi sono in relazione happened-before ($e \rightarrow e'$) se è vero uno dei seguenti casi:
   1. $\exists\ p_i \mid \text{e} \rightarrow_i e'$
   2. $e = \text{send}(m) \wedge e' = \text{receive}(m)$
   3. $\exists$ e, e', e" $\mid (e \rightarrow e'') \wedge (e'' \rightarrow e')$

1.1.1 Clock logico scalare

Il clock logico scalare L_i è associato a ciascun singolo processo p_i e assegna un singolo valore intero monotonicamente crescente ad ogni evento. Questo tipo di clock fornisce un ordinamento parziale degli eventi all'interno di ciascun processo, non tiene infatti conto degli eventi relativi agli altri processi nel sistema. Utilizzando come meccanismo di sincronizzazione i clock logici scalari non è quindi possibile stabilire se due eventi sono concorrenti o meno. L' implementazione avviene tramite Algoritmo di Lamport.

Lamport's Algorithm

Forall process p_i, L_i=0
Before internal event p_i **set** $L_i \leftarrow L_i+1$
If p_i sends message to p_j:
  $L_i \leftarrow L_i + 1$
  $t \leftarrow L_i$
  timestamp(m) $\leftarrow t$
  send(m)
If p_j receives m with timestamp(m)=t
  $L_j \leftarrow \max(t, L_j)$
  $L_j \leftarrow L_j + 1$
  receive(m)

1.1.2 Clock logico vettoriale

Il clock logico vettoriale V_i, implementato come un vettore di N interi, viene associato ad ogni processo. Ogni componente del vettore rappresenta il tempo logico del rispettivo processo, consentendo di stabilire la possibile concorrenza tra due eventi all'interno del sistema.

Vector Logical Clocks Update

Forall process p_i, $V_i[k] = 0 \,\forall\, k = 1, 2, \ldots, N$
Before internal event p_i **set** $V_i[i] \leftarrow V_i[i] + 1$
If p_i sends message to p_j:
  $V_i[i] \leftarrow V_i[i] + 1$
  $t \leftarrow V_i$
  timestamp(m)$\leftarrow t$
  send(m)
If p_j receives m with timestamp(m)=t
  $L_j[k] \leftarrow \max(t[k], V_j[k]) \,\forall\, k = 1, 2, \ldots, N$
  $V_j[j] \leftarrow V_j[j] + 1$
  receive(m)

4 Condizioni necessarie per l'utilizzo degli algoritmi di Multicast in un Datastore Key-Value

4.1 Identificazione unica dei nodi

È essenziale che ogni nodo nel sistema distribuito sia distintamente identificato tramite un indice univoco. Questo permette a ciascun processo di procedere senza ambiguità nell'invio di messaggi in modalità multicast.

4.2 Comunicazione

Per garantire il corretto funzionamento delle operazioni assumiamo una comunicazione affidabile, possiamo quindi contare sul fatto che i messaggi inviati tra i nodi del sistema vengano consegnati correttamente, senza perdite o alterazioni.

4.3 FIFO ordered

Per assicurarci il funzionamento delle operazioni assumiamo una comunicazione FIFO ordered, ovvero che i messaggi inviati da un processo p_i ad un altro processo p_j sono ricevuti da p_j nello stesso ordine in cui p_i li ha inviati escludendo possibili ritardi di rete. Per assicurare questa proprietà stabiliamo un campo del messaggio che identifica il suo numero di sequenza, e serviamo solamente il messaggio con il numero che ci aspettiamo di ricevere.

4.4 Assenza di guasti

In questa analisi, poniamo come ipotesi che durante l'esecuzione delle operazioni considerate non si verifichi nessun guasto. Definiamo quindi un processo corretto come una procedura che non manifesta mai comportamenti difettosi.

5. *Descrizione del funzionamento degli algoritmi di Multicast in un Datastore Key-Value*

Il Datastore distribuito è modellato come un set di processi $\Pi = \{p_1, \ldots, p_n\}$ che interagiscono tra di loro mantenendosi consistenti grazie all'invio e alla ricezione di messaggi multicast tramite canali di comunicazioni. Il Datastore è basato sul paradigma di archiviazione di dati Key-Value, in cui i record vengono archiviati e recuperati utilizzando una chiave che lo identifica in modo univoco. Ogni processo possiede il proprio Datastore Key-Value. Distinguiamo tre diversi eventi associati allo scambio di messaggi: *get, put, delete*. L'operazione di get recupera all'interno del datastore il valore univoco associato ad una determinata chiave, l'operazione di put permette l'inserimento di una nuova coppia Key-Value all'interno del datastore, mentre l'operazione di Delete permette la completa rimozione di un elemento dal datastore. Ogni evento può essere eseguito attraverso algoritmo di Multicast Totalmente Ordinato, seguendo quindi principi di consistenza sequenziale oppure attraverso algoritmo di Multicast Causalmente Ordinato seguendo principi di consistenza causale, a seconda delle esigenze specifiche del sistema.

5.1 Multicast Totalmente Ordinato

Il problema del Multicast Totalmente Ordinato è definito da quattro proprietà che devono essere rispettate: validità, accordo, integrità e ordine totale. Queste proprietà assumono nello specifico contesto i seguenti significati:

- Validità: Se un processo riceve un messaggio m, allora tutti i processi corretti appartenenti all'insieme di destinazione *Dest(m)* riceveranno prima o poi m.
- Accordo uniforme: se un processo consegna un messaggio m, allora tutti i processi corretti all'interno dell'insieme *Dest(m)* riceveranno m.
- Integrità uniforme: ogni messaggio m viene ricevuto da ogni processo p al massimo una volta, e solo se m è stato precedentemente inviato in multicast da *sender(m)* e p è un processo incluso nell'insieme di destinazione *Dest(m)*.
- Ordine totale Uniforme: se un processo p e un processo q consegnano rispettivamente il messaggio m ed m', allora p consegnerà m prima di m' se e solo se q consegnerà m prima di m'.

Nell'algoritmo di Multicast Totalmente Ordinato, basato sull'uso dei clock scalari, ogni processo mantiene una coda di messaggi totalmente ordinata in base al timestamp del messaggio ed un datastore Key-Value che ospiterà i record inseriti dal Client, quando il Client decide di inserire un elemento all'interno del datastore, sarà inviato un messaggio ad un server p_i all'interno della lista dei server disponibili, il server riceverà il messaggio, che chiamiamo m_i, ed andrà ad impostare il timestamp di m_i pari al suo clock scalare. Ora, tramite comunicazione multicast, invia il messaggio m_i a tutti i server facente parti del gruppo di multicast, compreso sè stesso. A questo punto ogni server che riceve m_i lo inserisce nella sua coda ed invia, sempre in multicast, un *acknowledgement*, ovvero un messaggio per informare gli altri server che la ricezione è andata a buon fine. A questo punto ogni server procede con la consegna del messaggio all'applicazione se:

1. Il messaggio è in testa alla coda del processo.
2. Il server ha ricevuto un messaggio di acknowledgement, relativo a m_i, da parte di tutti i server facenti parte del gruppo di multicast.
3. Per ogni processo p_k, c'è un messaggio msg_k, all'interno della propria coda con timestamp maggiore di quello di m_i.

Nei sistemi distribuiti la scalabilità riveste un'importanza cruciale, è quindi essenziale affrontare la questione della scalabilità a livello architetturale del sistema. Questo algoritmo ha costo di comunicazione non indifferente, notiamo infatti che abbiamo N processi, e che per ogni messaggio che un processo riceve viene inviato un ack in multicast, generando N^2 ack. Questo costo di comunicazione porta a concludere che il Multicast Totalmente Ordinato ha una scalabilità limitata.

5.2 Multicast Causalmente Ordinato

Nel protocollo di Multicast Causalmente Ordinato, rispettiamo la proprietà di Validità, Accordo uniforme e integrità uniforme, mentre non rispettiamo l'ordine totale uniforme, questo perchè in questo caso tutti i messaggi devono essere consegnati in un ordine consistente con la causalità. Supponiamo che un gruppo di processi P comunichi utilizzando Multicast Causalmente Ordinato, ogni processo $p \in P$ riceverà i messaggi che gli vengono inviati, li rallenterà se necessario, e poi li consegnerà rispettando le relazioni di causa-effetto. Analizziamo queste relazioni:

1. Un evento di read seguito da un evento di write sullo stesso processo: write è (potenzialmente) causalmente correlata con read.
2. Evento di write di un dato seguita da evento di read dello stesso dato su processi diversi: read è (potenzialmente) causalmente correlata con l'evento di write.
3. Si applica la proprietà transitiva: se P1 scrive x e P2 legge x e usa il valore letto per scrivere y, l'evento di read di x e l'evento di scrittura di y sono causalmente correlate tra di loro.

Notiamo che se due processi scrivono simultaneamente, le due write non sono causalmente correlate ma si definiscono write concorrenti. Ogni processo mantiene il proprio Datastore Key-Value e una coda dei messaggi ricevuti, quando il processo p_i invia un messaggio, allega il proprio clock al messaggio impostando il timestamp. Quando p_j riceve m da p_i lo pone in un coda di attesa, ne ritarda la consegna finchè:

1. $t(m)[i] = V_j[i] + 1$, ovvero m è il messaggio successivo che p_j si aspetta da p_i

2. $t(m)[k] \leqslant V_j[k]\ \forall\, k \neq i$, per ogni altro processo p_k, p_j ha visto almeno gli stessi messaggi visti da p_i

6 Implementazione degli algoritmi di Multicast Totalmente e Causalmente Ordinato in un data-store Key-Value

Definiamo la scelta delle tecnologie e l'architettura di sistema.

6.1 Scelta delle tecnologie

Linguaggio di Programmazione. Il linguaggio di programmazione scelto è Go, un linguaggio open-source sviluppato da Google. Uno dei motivi principali dietro la scelta di Go è il supporto nativo alla concorrenza, infatti la presenza delle Goroutine, thread leggeri gestiti dal runtime di Go, semplifica la creazione di codice concorrente, evitando la complessità associata ai thread tradizionali. Questa caratteristica rende Go adatto alla realizzazione di sistemi che necessitano di alta scalabilità come i sistemi distribuiti. Altra motivazione risiede nelle sue prestazioni, infatti la natura compilata e l'efficace runtime di Go lo rendono un linguaggio ad alte prestazioni specialmente in scenari che privilegiano la velocità come servizi di rete.

Cloud Provider. Il Cloud Provider usato è Amazon Web Services (AWS), la scelta dell'uso di AWS per l'implementazione di algoritmi di consistenza è motivata dalla sua affidabilità, scalabilità e facilità di gestione. La numerosa presenza inoltre dei data center AWS su scala globale garantisce una bassa latenza e contemporaneamente una alta disponibilità.

6.2 Architettura Generale

Nell'implementazione del Datastore Key-Value utilizziamo un'istanza EC2, ovvero un virtual server nella terminologia di AWS, per ospitare i vari server del sistema. Attraverso l'utilizzo di Docker, che fornisce portabilità, consistenza e scalabilità per la distribuzione di applicazioni, creiamo i vari container che ospiteranno i nostri server e i nostri client. L'uso dei container permette alle applicazioni di essere più rapidamente distribuite e scalate. Per avviare i container utilizziamo Docker Compose, grazie al quale possiamo gestire applicazioni multi-container con l'uso di file YAML.

6.3 Test

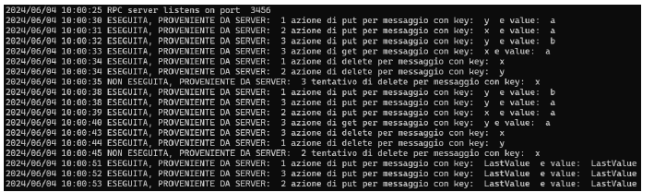
Per verificare che gli algoritmi rispettassero le garanzie delle rispettive consistenze sono stati eseguiti vari test che ora analizzeremo.

6.3.1 Test per consistenza sequenziale

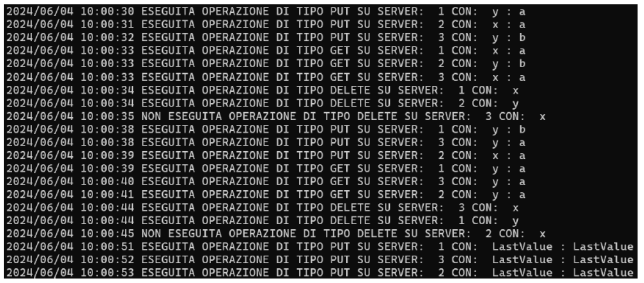
Vogliamo verificare con questo test che tutti i server rispettino la consistenza sequenziale, ovvero quindi che mantengano l'illusione di una singola copia e che sia possibile combinare tutte le richieste in un dato ordine sequenziale. Ricordo che per effettuare i test sono stati instanziati un numero di Client pari al numero di Server, ed ogni Server era quindi dedicato alla esecuzione delle richieste di un determinato Client.

P0:	W(y)a	R(x)	D(x)	R(x)	W(y)b	R(y)	D(y)	**R(y)**
P1:	W(x)a	**R(y)**	D(y)	**R(y)**	W(x)a	R(y)	D(x)	**R(y)**
P2:	W(y)b	R(x)	D(x)	**R(y)**	W(y)a	R(y)	D(x)	R(x)

Vediamo il comportamento dei server:



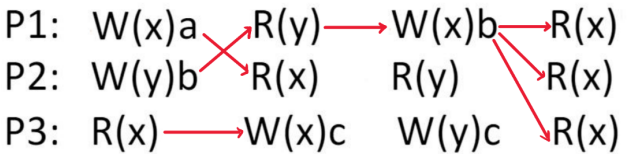
Output del Client:



Come vediamo l'illusione della singola copia è mantenuta, e possiamo trovare un interleaving che soddisfi l'ordine sequenziale delle operazioni.

6.3.2 Test per consistenza causale

Vogliamo dimostrare che tutti i server rispettino la consistenza causale, ovvero quindi che le operazioni in relazione di causa-effetto vengano riconosciute ed eseguite da tutti i server nello stesso ordine. Anche in questo caso ogni Client invia le richieste ad un singolo server. Sono state evidenziate all'interno del test le relazioni di causa-effetto.



Vediamo ora l'output dei vari server:

```
2024/06/04 11:56:41 RPC server listens on port  1234
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  1 azione di put per messaggio con key:  x  e value:  a
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  2 azione di put per messaggio con key:  y  e value:  b
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  3 azione di put per messaggio con key:  x  e value:  c
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  1 azione di get per messaggio con key:  y e value:  b
2024/06/04 11:56:57 ESEGUITA, PROVENIENTE DA SERVER:  3 azione di put per messaggio con key:  y  e value:  c
2024/06/04 11:56:57 ESEGUITA, PROVENIENTE DA SERVER:  1 azione di put per messaggio con key:  x  e value:  b
2024/06/04 11:56:59 ESEGUITA, PROVENIENTE DA SERVER:  1 azione di get per messaggio con key:  x e value:  b
```

```
2024/06/04 11:56:42 RPC server listens on port  2345
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  1 azione di put per messaggio con key:  x  e value:  a
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  2 azione di put per messaggio con key:  y  e value:  b
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  3 azione di put per messaggio con key:  x  e value:  c
2024/06/04 11:56:57 ESEGUITA, PROVENIENTE DA SERVER:  2 azione di get per messaggio con key:  x e value:  c
2024/06/04 11:56:57 ESEGUITA, PROVENIENTE DA SERVER:  3 azione di put per messaggio con key:  y  e value:  c
2024/06/04 11:56:57 ESEGUITA, PROVENIENTE DA SERVER:  1 azione di put per messaggio con key:  x  e value:  b
2024/06/04 11:56:58 ESEGUITA, PROVENIENTE DA SERVER:  2 azione di get per messaggio con key:  y e value:  c
2024/06/04 11:57:00 ESEGUITA, PROVENIENTE DA SERVER:  2 azione di get per messaggio con key:  x e value:  b
```

```
2024/06/04 11:56:42 RPC server listens on port  3456
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  1 azione di put per messaggio con key:  x  e value:  a
2024/06/04 11:56:56 ESEGUITA, PROVENIENTE DA SERVER:  2 azione di put per messaggio con key:  y  e value:  b
2024/06/04 11:56:57 ESEGUITA, PROVENIENTE DA SERVER:  3 azione di get per messaggio con key:  x e value:  a
2024/06/04 11:56:57 ESEGUITA, PROVENIENTE DA SERVER:  3 azione di put per messaggio con key:  x  e value:  c
2024/06/04 11:56:57 ESEGUITA, PROVENIENTE DA SERVER:  1 azione di put per messaggio con key:  x  e value:  b
2024/06/04 11:56:58 ESEGUITA, PROVENIENTE DA SERVER:  3 azione di put per messaggio con key:  y  e value:  c
2024/06/04 11:56:59 ESEGUITA, PROVENIENTE DA SERVER:  3 azione di get per messaggio con key:  x e value:  b
```

Output del Client:

```
2024/06/04 11:56:56 ESEGUITA OPERAZIONE DI TIPO PUT SU SERVER:  1 CON:  x : a
2024/06/04 11:56:56 ESEGUITA OPERAZIONE DI TIPO PUT SU SERVER:  2 CON:  y : b
2024/06/04 11:56:56 ESEGUITA OPERAZIONE DI TIPO GET SU SERVER:  1 CON:  y : b
2024/06/04 11:56:57 ESEGUITA OPERAZIONE DI TIPO GET SU SERVER:  2 CON:  x : c
2024/06/04 11:56:57 ESEGUITA OPERAZIONE DI TIPO GET SU SERVER:  3 CON:  x : a
2024/06/04 11:56:57 ESEGUITA OPERAZIONE DI TIPO PUT SU SERVER:  3 CON:  x : c
2024/06/04 11:56:57 ESEGUITA OPERAZIONE DI TIPO PUT SU SERVER:  1 CON:  x : b
2024/06/04 11:56:58 ESEGUITA OPERAZIONE DI TIPO PUT SU SERVER:  3 CON:  y : c
2024/06/04 11:56:58 ESEGUITA OPERAZIONE DI TIPO GET SU SERVER:  2 CON:  y : c
2024/06/04 11:56:59 ESEGUITA OPERAZIONE DI TIPO GET SU SERVER:  3 CON:  x : b
2024/06/04 11:56:59 ESEGUITA OPERAZIONE DI TIPO GET SU SERVER:  1 CON:  x : b
2024/06/04 11:57:00 ESEGUITA OPERAZIONE DI TIPO GET SU SERVER:  2 CON:  x : b
```

Come vediamo tutti i i server riconoscono la relazione di causa-effetto ed eseguono nel corretto ordine le operazioni.

7 Risultati finali

L'utilizzo della consistenza sequenziale, e quindi l'implementazione di un algoritmo di Multicast Totalmente Ordinato, comporta un overhead dovuto al sistema di acknowledgement non presente invece nella consistenza causale e quindi nell'algoritmo di Multicast Causalmente Ordinato. Per concludere, utilizzare la consistenza sequenziale porta a prestazioni minori ma contemporaneamente permette di mantenere l'illusione di avere una singola copia, che viene invece persa con l'uso della consistenza causale. La scelta tra le due dipende quindi dalle necessità del singolo sistema.

References

[1] Défago X., Schiper A., & Urbán P. (Anno: 2000). Totally Ordered Broadcast and Multicast Algorithms: A Comprehensive Survey.

[2] Lamport L. (Anno: 1978). Time, Clocks, and the Ordering of Events in a Distributed System. Communications of the ACM, 21(7), 558-565.

[3] Rodrigues L., Verìssimo P., Casimiro A. (Anno: 1995). Priority-Based Totally Ordered Multicast.

[4] Birman K, (Anno: 1990). Fast Causal Multicast.