Consistency and Replication

Bartolomeo Lombardi
Andrea Segalini
Amerigo Mancino

10 novembre 2015

Presentazione

In questa breve presentazione, parleremo di replicazione dei dati e della conseguente necessità di mantenere questi dati coerenti fra loro.
Il nostro percorso si articola in 4 fasi principali:

1. Presentazione e definizione di alcuni livelli di coerenza, spiegati anche attraverso uno pseudocodice che simula il gioco del Baseball;
2. Illustazione del sistema Amazon Dynamo, con particolare enfasi ai livelli di coerenza implementati;
3. Illustazione del sistema COPS e del particolare livello di coerenza implementato in questo caso;
4. Conclusioni e considerazioni finali.

Replicated Data Consistency Explained Through Baseball

Articolo di riferimento:

Terry, Doug.
Replicated Data Consistency Explained Through Baseball
Communications of the ACM 56.12 (2013): 82-89, (2013, December)

Replica di dati nel cloud

- Sistemi di storage replicano i dati su più macchine per la resistenza ai guasti e per migliorare le performance.
- Si utilizza la geo-replication per resistere ad una eventuale perdita di datacenter interi.
- I datacenter vengono distribuiti a livello globale in più continenti per garantire che i dati siano vicini a più basi di client.

Sistemi di storage popolari

Di seguito presentiamo alcuni esempi di sistemi e il loro grado di coerenza implementato:

- Amazon S3 – eventual consistency
- Amazon Simple DB – eventual o strong
- Google App Engine – strong o eventual
- Yahoo! PNUTS – eventual o strong
- Windows Azure Storage – strong o eventual

In questa presentazione:

- Amazon Dynamo - strong o eventual
- COPS - casual+ consistency

Teorema CAP (Teorema di Brewer)

Definizione

Il teorema CAP afferma che è impossibile per un sistema informatico distribuito fornire simultaneamente tutte e tre le seguenti garanzie:

- **Consistency**
  Tutti i nodi vedono gli stessi dati nello stesso momento
- **Availability**
  La garanzia che ogni richiesta riceva una risposta su ciò che sia riuscito o fallito
- **Partition Tolerance**
  Il sistema continua a funzionare nonostante arbitrarie perdite di messaggi

Secondo il teorema, un sistema distribuito è in grado di soddisfare al massimo due di queste garanzie allo stesso tempo, ma mai tutte e tre.

Alcuni tipi di coerenza I

- **Strong Consistency**: garantisce che l'operazione di lettura ritorni il valore dell'ultima scrittura.
- **Eventual Consistency**: è la più debole delle altre, poiché permette di avere come valore di ritorno il più grande insieme di possibilità.

Alcuni tipi di coerenza II

- **Consistent Prefix**: il lettore vede una qualunque versione del datastore che è esistita in qualche momento del passato.
- **Bounded Staleness**: assicura di ritornare il valore che non sia più vecchio di un lasso di tempo T

Alcuni tipi di coerenza III

- **Monotonic Reads**: se il client effettua la lettura di un dato ed ottiene un certo valore X è garantito che se rilegge non riceverà un dato che è stato scritto prima di X.
- **Read My Writes**: abbiamo coerenza forte rispetto alle modifiche effettuate dal client stesso mentre eventual per tutte le altre.

Coerenze a confronto

Guarantee	Consistency	Performance	Availability
Strong Consistency	excellent	poor	poor
Eventual Consistency	poor	excellent	excellent
Consistent Prefix	okay	good	excellent
Bounded Staleness	good	okay	poor
Monotonic Reads	okay	good	good
Read My Writes	okay	okay	okay

Il baseball in pseudocodice

```
Write ("visitors", 0);
Write ("home", 0);
  for inning = 1 .. 9
    outs = 0;
    while outs < 3
      visiting player bats;
      for each run scored
        score = Read ("visitors");
        Write ("visitors", score + 1);
    outs = 0;
    while outs < 3
      home player bats;
      for each run scored
        score = Read ("home");
        Write ("home", score + 1);
end game;
```

Un esempio

Team	1	2	3	4	5	6	7	8	9	RUNS
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

Letture di punteggi possibili per ogni tipo di coerenza	
Strong Consistency	2-5
Eventual Consistency	0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5
Consistent Prefix	0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5
Bounded Staleness	un inning indietro: 2-3, 2-4, 2-5
Monotonic Reads	dopo aver letto 1-3: 1-3, 1-4, 1-5, 2-3, 2-4, 2-5
Read My Writes	per chi scrive: 2-5 altri: eventual consistency

Soggetti del baseball

Nel gioco del baseball vi sono differenti soggetti con diversi ruoli, ognuno di essi necessita di un modello di coerenza diverso:

- Segnapunti
- Arbitro
- Radiocronista
- Giornalista sportivo
- Addetto alle statistiche
- Osservatore

Segnapunti

```
score = Read ("visitors");
Write ("visitors", score + 1);
```

Coerenza desiderata

- Strong consistency è superflua, il segnapunti è l'unico ad effettuare delle write.
- Read My Writes è sufficiente.

Segnapunti

```
score = Read ("visitors");
Write ("visitors", score + 1);
```

Coerenza desiderata

- Strong consistency è superflua, il segnapunti è l'unico ad effettuare delle write.
- Read My Writes è sufficiente.

Segnapunti

```
score = Read ("visitors");
Write ("visitors", score + 1);
```

Coerenza desiderata

- Strong consistency è superflua, il segnapunti è l'unico ad effettuare delle write.
- Read My Writes è sufficiente.

Arbitro

```
if first half of 9th inning complete then
  vScore = Read ("visitors");
  hScore = Read ("home");
  if vScore < hScore
    end game;
```

Coerenza desiderata

L'arbitro necessita della Strong Consistency per determinare la squadra vincitrice e concludere la partita.

Arbitro

```
if first half of 9th inning complete then
  vScore = Read ("visitors");
  hScore = Read ("home");
  if vScore < hScore
    end game;
```

Coerenza desiderata

L'arbitro necessita della Strong Consistency per determinare la squadra vincitrice e concludere la partita.

Radiocronista

```
do {
    vScore = Read ("visitors");
    hScore = Read ("home");
    report vScore and hScore;
    sleep (30 minutes);
}
```

Coerenza desiderata

- Consistent Prefix non è sufficiente, la lettura successiva può avvenire su un nodo non sincronizzato.
  Esempio: dopo aver letto lo score 2-5 è possibile leggere il punteggio 1-3.
- Monotonic Reads o Bounded Staleness è necessaria.

Radiocronista

```
do {
    vScore = Read ("visitors");
    hScore = Read ("home");
    report vScore and hScore;
    sleep (30 minutes);
}
```

Coerenza desiderata

- Consistent Prefix non è sufficiente, la lettura successiva può avvenire su un nodo non sincronizzato.
  Esempio: dopo aver letto lo score 2-5 è possibile leggere il punteggio 1-3.
- Monotonic Reads o Bounded Staleness è necessaria.

Radiocronista

```
do {
    vScore = Read ("visitors");
    hScore = Read ("home");
    report vScore and hScore;
    sleep (30 minutes);
}
```

Coerenza desiderata

- Consistent Prefix non è sufficiente, la lettura successiva può avvenire su un nodo non sincronizzato.
  Esempio: dopo aver letto lo score 2-5 è possibile leggere il punteggio 1-3.
- Monotonic Reads o Bounded Staleness è necessaria.

Giornalista sportivo

```
While not end of game {
  drink beer;
  smoke cigar;
}
go out to dinner;
vScore = Read ("visitors");
hScore = Read ("home");
write article;
```

Coerenza desiderata

- Eventual Consistency ?
- Bounded Staleness garantisce la sicurezza sul risultato finale.

Giornalista sportivo

```
While not end of game {
  drink beer;
  smoke cigar;
}
go out to dinner;
vScore = Read ("visitors");
hScore = Read ("home");
write article;
```

Coerenza desiderata

- Eventual Consistency ?
- Bounded Staleness garantisce la sicurezza sul risultato finale.

Giornalista sportivo

```
While not end of game {
  drink beer;
  smoke cigar;
}
go out to dinner;
vScore = Read ("visitors");
hScore = Read ("home");
write article;
```

Coerenza desiderata

- Eventual Consistency ?
- Bounded Staleness garantisce la sicurezza sul risultato finale.

Addetto alle statistiche

```
Wait for end of game;
score = Read ("home");
stat = Read ("season-runs");
Write ("season-runs", stat + score);
```

Coerenza desiderata

Addetto alle statistiche necessita di due dati:

- Punteggio ultima partita (home):
  - Strong Consistency: se effettuata il giorno stesso (lasso di tempo breve)
  - Bounded Staleness: se diversi giorni dopo
- Punteggi accumulati (season-runs):
  - Read My Writes: se è l'unico ad effettuare le scritture
  - Strong Consistency: altrimenti

Addetto alle statistiche

```
Wait for end of game;
score = Read ("home");
stat = Read ("season-runs");
Write ("season-runs", stat + score);
```

Coerenza desiderata

Addetto alle statistiche necessita di due dati:

- Punteggio ultima partita (home):
  - Strong Consistency: se effettuata il giorno stesso (lasso di tempo breve)
  - Bounded Staleness: se diversi giorni dopo
- Punteggi accumulati (season-runs):
  - Read My Writes: se è l'unico ad effettuare le scritture
  - Strong Consistency: altrimenti

Addetto alle statistiche

```
Wait for end of game;
score = Read ("home");
stat = Read ("season-runs");
Write ("season-runs", stat + score);
```

Coerenza desiderata

Addetto alle statistiche necessita di due dati:

- Punteggio ultima partita (home):
  - Strong Consistency: se effettuata il giorno stesso (lasso di tempo breve)
  - Bounded Staleness: se diversi giorni dopo
- Punteggi accumulati (season-runs):
  - Read My Writes: se è l'unico ad effettuare le scritture
  - Strong Consistency: altrimenti

Osservatore

```
do {
    stat = Read ("season-runs");
    discuss stats with friends;
    sleep (1 day);
}
```

Coerenza desiderata

Eventual Consistency è sufficiente, le statistiche vengono aggiornate ogni giorno

Osservatore

```
do {
    stat = Read ("season-runs");
    discuss stats with friends;
    sleep (1 day);
}
```

Coerenza desiderata

Eventual Consistency è sufficiente, le statistiche vengono aggiornate ogni giorno

Dynamo: Amazon's Highly Available Key-value Store

Articolo di riferimento:

DeCandia, G., Hastorum, D., Jampani, M., Kakalapati, G., Lakshman A., Pilchin, A., Sivasubramanian, S., Vosshall, P. & Vogels, W.
Dynamo: Amazon's Highly Available Key-value Store
ACM SIGOPS Operating Systems Review (Vol. 41, No. 6, pp. 205-220), (2007, October)

Dynamo

Cos'è Dynamo

Dynamo è un sistema di key-value storage distribuito.

Obiettivi

- "Always on" experience.
- Scalabilità.
- Alte performance - bassa latenza.
- Eventual consistency.

In realtà il sistema è altamente configurabile, è possibile migliorare alcuni aspetti a scapito di altri...

Architettura del Sistema

Scomponiamo il sistema in base alle tecniche utilizzate:

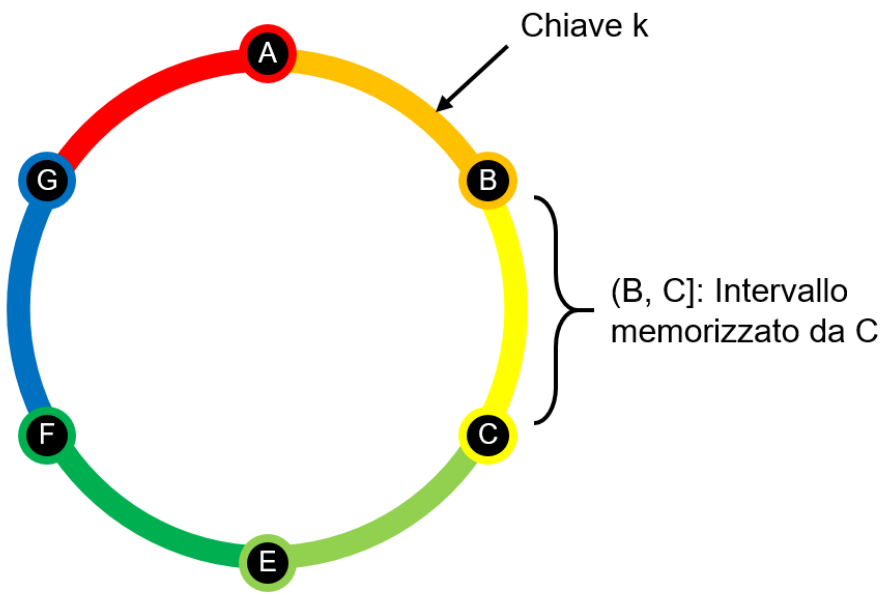
Partizionamento e replicazione	Consistent hashing
Versionamento	Vector clock
Coerenza operazioni	Quorum likeness
Gestione fallimenti	Sloppy quorum, hinted handoff, Merkle tree
Membership e rilevamento fallimenti	Gossip-based protocol,

Partizionamento (I)

- Si usa la tecnica del consistent hashing.

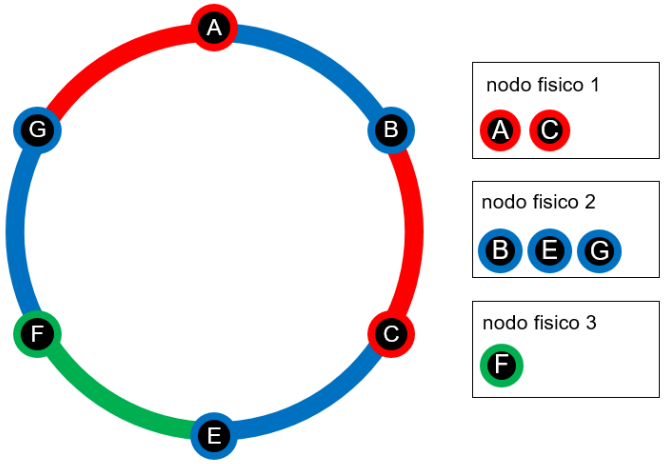
Consistent hashing

- Tabella hash richiusa ad anello.
- Nodi posizionati casualmente sull'anello a formare intervalli.
- Ogni nodo memorizza le chiavi nell'intervallo tra lui e il precedente.



Partizionamento (II)

Dynamo propone una variante del consistent hashing, utilizzo di nodi virtuali (tokens).



Vantaggi

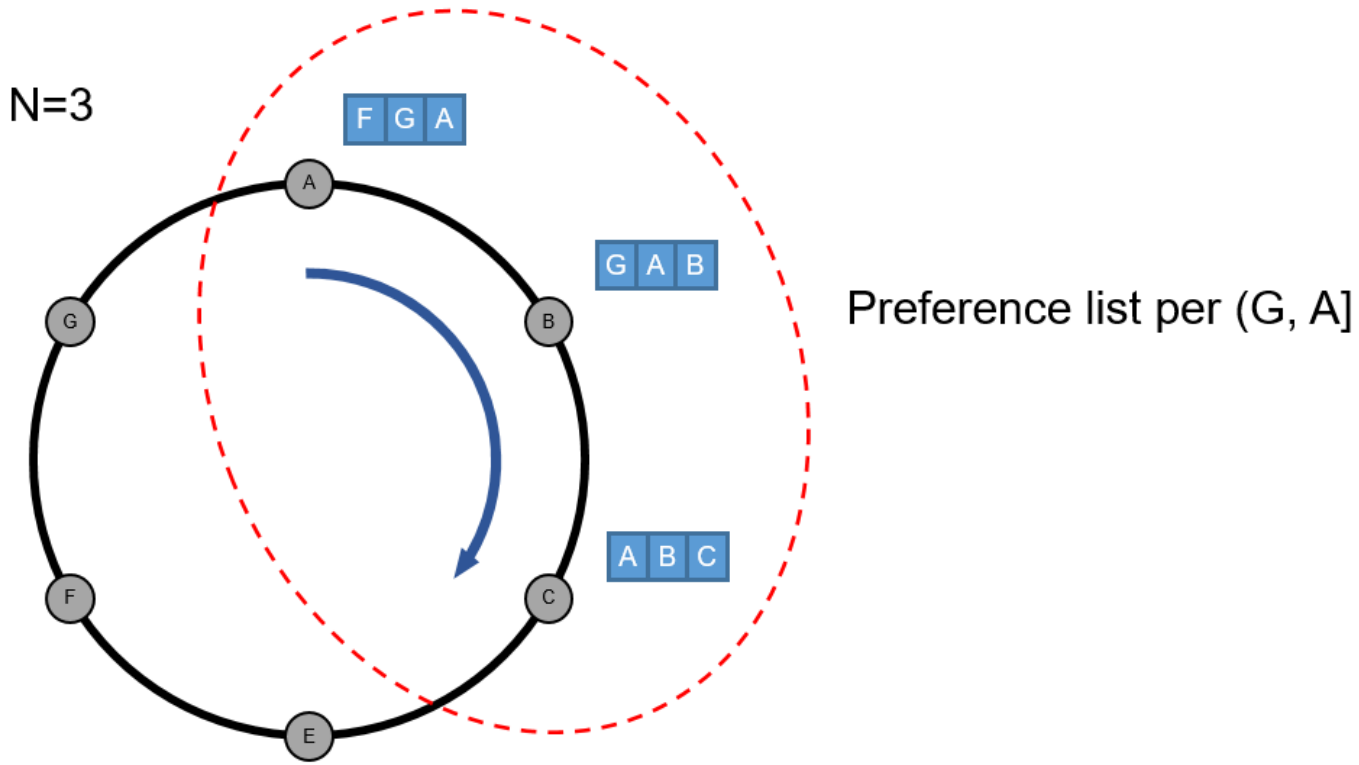
- Dati distribuiti uniformemente.
- Aggiungere o togliere nodi influenza solo il "successore".
- Bilanciamento del carico a seguito di un fallimento.
- Si tiene conto dell'eterogeneità delle macchine.

$\longrightarrow$ In poche parole si ottiene una forte scalabilità.

Replicazione

Ogni nodo effettua una replica sugli N nodi successivi (senso orario).

- L'insieme dei nodi che ospitano il medesimo intervallo di chiavi è detto preference list.
- I nodi virtuali sono saltati lungo l'anello.
- L'aggiornamente delle repliche è asincrono.



Versionamento (I)

- Ogni modifica viene trattata dal sistema come un nuovo oggetto.
- Si utilizzano i vector clock (VC) per tenere traccia delle versioni:

$$\left[< \text{nodo}, \text{counter} >< n_2, c_2 > \ldots < n_m, c_m > \right]$$

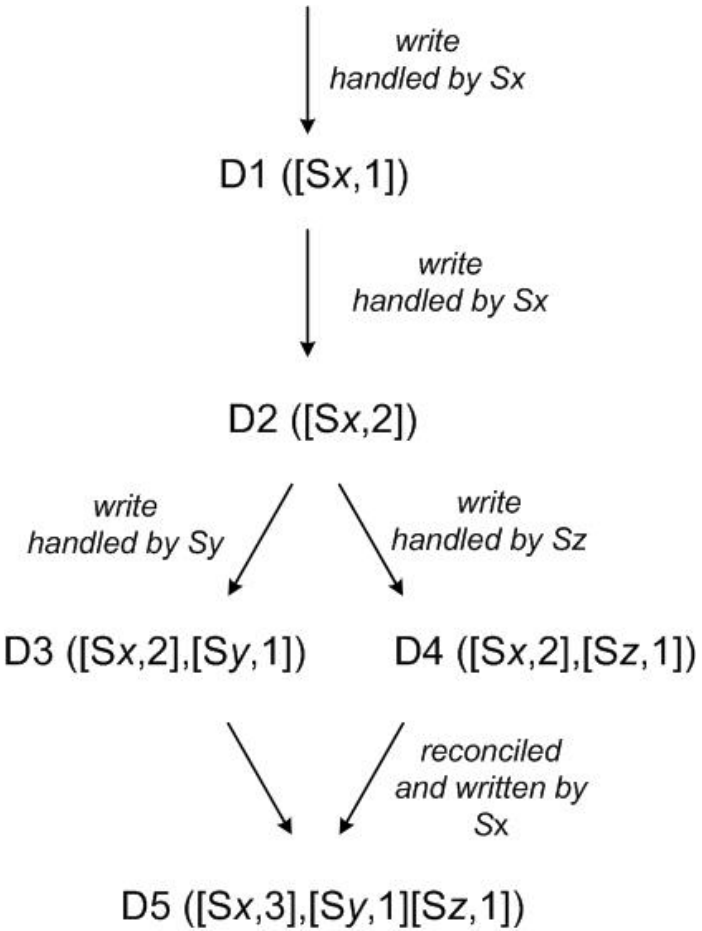
Definizione

Siano x e y due versioni del medesimo oggetto, denotiamo con $\text{VC}(x)_z$ il vector clock di x relativo al nodo z.

$$\text{VC}(x) < \text{VC}(y) \iff \left(\forall z : \text{VC}(x)_z \leq \text{VC}(y)_z\right) \wedge \left(\exists z' : \text{VC}(x)_{z'} < \text{VC}(y)_{z'}\right)$$

Versionamento (II)

Un esempio:



1. D1 - Scrittura gestita da S_x.
2. D2 - Scrittura gestita da S_x.
3. S_x fallisce dopo aver propagato.
4. D3 - Scrittura gestita da S_y.
5. S_y fallisce prima di aver propagato.
6. D4 - Scrittura gestita da S_z.
7. S_y torna online.
8. D3 e D4 sono concorrenti.

Versionamento (III)

Si ottiene un ordinamento parziale sulle versioni:

- Se $\mathrm{VC}(x) < \mathrm{VC}(y)$ allora la versione y può sovrascrivere x.
- Altrimenti abbiamo un branching delle versioni che il sistema non può riconciliare autonomamente.

Pro dell'uso dei vector clock

- Rispetto all'utilizzo di un timestamp logico si ottiene un ordinamento parziale tra le versioni.
- I vari branch di versionamento vengono tutti individuati e ritornati per una riconcialiazione da parte del client.

Contro

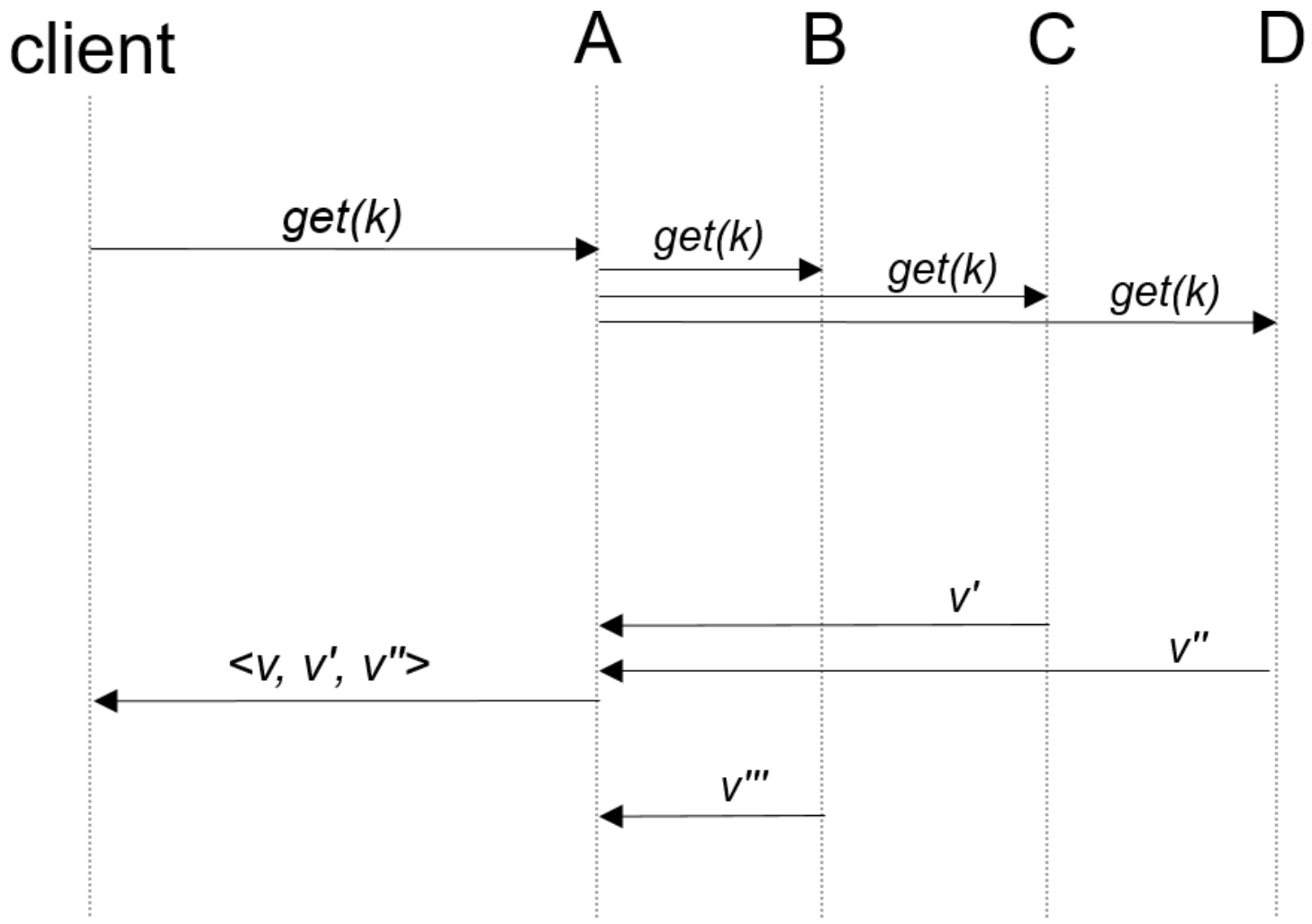
- La dimensione dei vector clock può crescere tanto.

Garantire la Coerenza

- Operazione di `get` e `put`:
  - `get(key) : <value, context>`
  - `put(key, value, context)`
- La coerenza è ottenuta mediante la tecnica del quorum:
  - W: ($W \leq N$) numero di nodi che contribuiscono alla scrittura.
  - R: ($R \leq N$) numero di nodi che contribuiscono alla lettura.

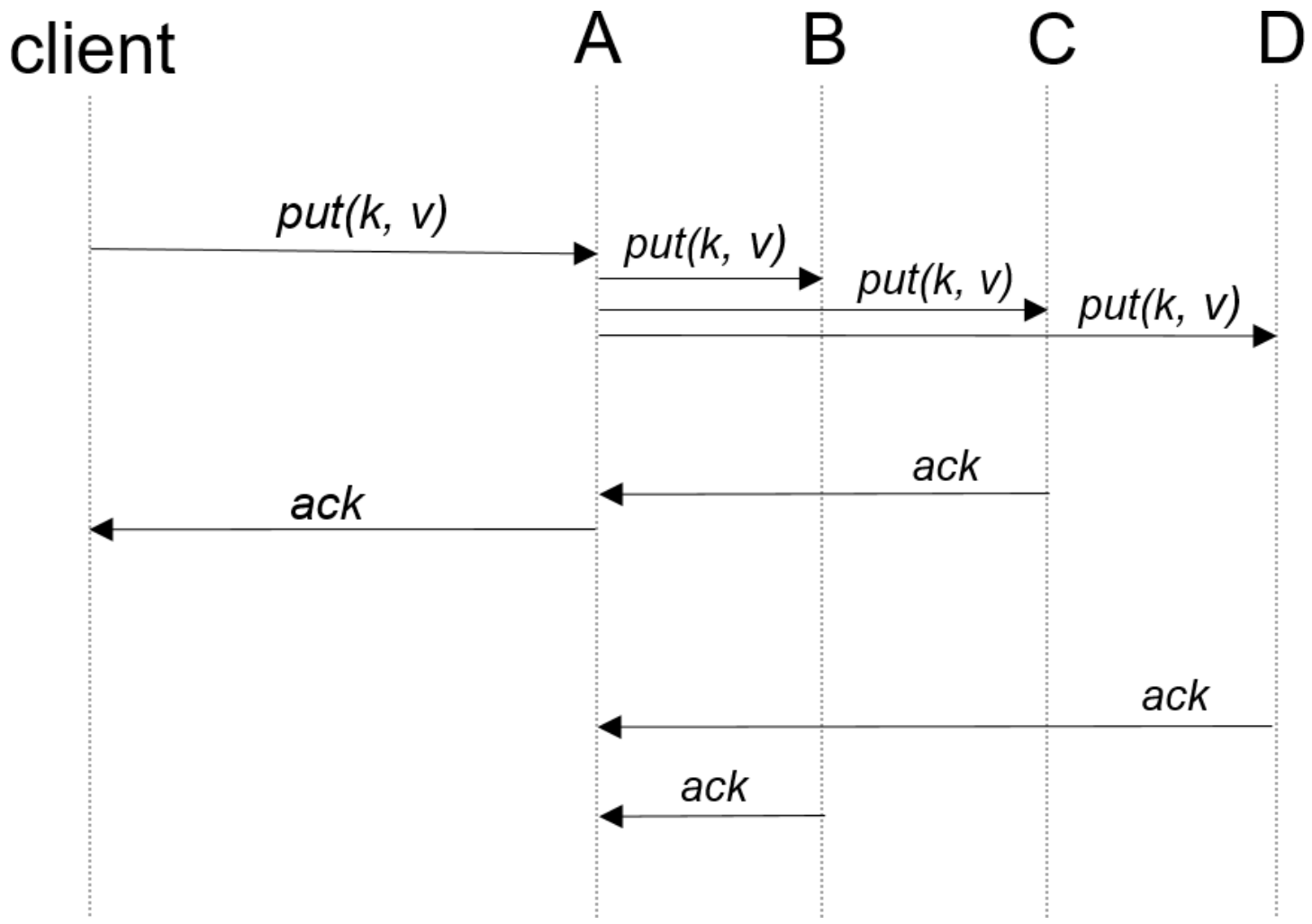
Operazione get

$N = 4, W = 2, R = 3$



Operazione put

$N = 4, W = 2, R = 3$



Proprietà di un quorum-like system

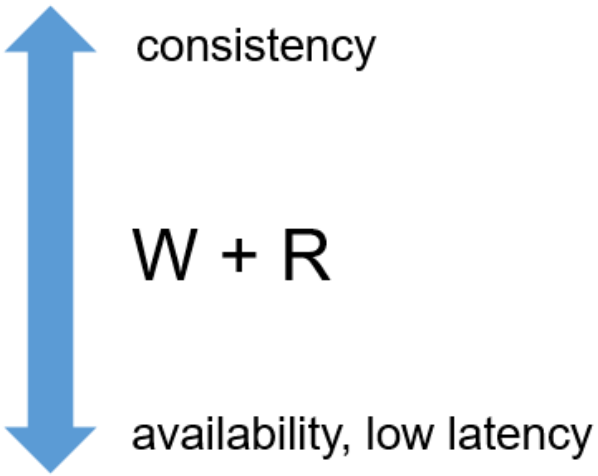
Se $R + W > N$

- in assenza di fallimenti abbiamo strong consistency.

Se $R + W \leq N$

- ottengo eventual consistency a vari livelli...
- $W = 1$: always writable (high availability).
- $R = 1$: always readable (high availability).

Rissumendo su W, R e N



Esempio

High performance read engine: Dynamo è pensato per essere un "always writable" data store, con $R = 1$ e $W = N$ diventa un sistema ideale per servire tante letture con poche e rare scritture.

Sloppy Quorum

Problema

Bastano pochi fallimenti nella "top N" della preference list per impedire il raggiungimento del quorum.

Si utilizza lo "sloppy quorum":

- Durante un'operazione si contattano i primi N nodi sani nella preference list (si scorre in senso orario l'anello).
- La replica che normalmente risiederebbe sul nodo fallito viene traferita al nodo supplente.

$\longrightarrow$ Quando i nodi falliti ritornano in servizio nasce un problema di incoerenza.

Gestione Fallimenti e Sincronia Repliche (I)

Hinted handoff

- La replica spedita al nodo "supplente" contiene un hint.
- Il nodo supplente si preoccupa di verificare periodicamente lo stato del nodo che sta sostituendo.
- Non appena possibile viene restituita la replica aggiornata al nodo fallito.

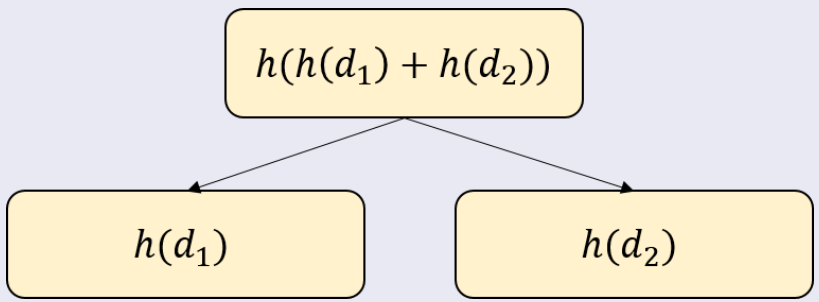
In questo modo si minimizza la finestra di vulnerabilità.

Gestione Fallimenti e Sincronia Repliche (II)

In caso di fallimento di un supplente è necessario un mezzo per stabilire se due repliche sono sincronizzate:

Merkle tree

- I Merkle tree sono alberi in cui ogni nodo padre è l'hash della somma dei figli.



- Consentono di identificare efficientemente differenze tra due alberi.
- Ogni nodo memorizza un Merkle tree per ogni intervallo che memorizza.

Membership e Rilevamento Fallimenti

Gestire l'appartenenza al sistema (membership)

- Non è presente una vista centralizzata dei membri appartenenti al sistema.
- Le informazioni sui membri sono scambiate mediante un protocollo gossip based.
- Presenza di nodi seed individuati "staticamente", notificati immediatamente dei cambiamenti nel sistema.

Rilevamento fallimenti

- Il concetto di fallimento è locale.
- Dunque non c'è una visione unica e coerente dei nodi falliti, necessario un monitoring periodico.

Concludendo su Dynamo

Vantaggi di Dynamo

- Scalabilità.
- Riconciliazione versioni gestita dal client.
- Coerenza, availability, latency, regolabili appositamente per un'istanza.

Svantaggi

- La riconciliazione a carico del client complica il codice delle applicazioni.
- Impossibile modificare dinamicamente i parametri N, W, R per adattarsi alla situazione del sistema alleggerendo il traffico di richieste interne.

Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS

Articolo di riferimento:

Wyatt Lloyd, M. J. Freedman, M. Kaminsky & Vogels, D. G. Andersen
Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS
Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, (ACM, 2011)

COPS

Cos'è COPS

COPS (Cluster of Order-Preserving Servers) è, come Dynamo, un key-value store distribuito in un'ampia area che implementa un modello di coerenza denomitato *causal+*.

Cos'è COPS-GT

COPS-GT è un'evoluzione di COPS che implementa le cosiddette le `get transaction`, capaci di ottenere una visione consistente di un insieme di chiavi.

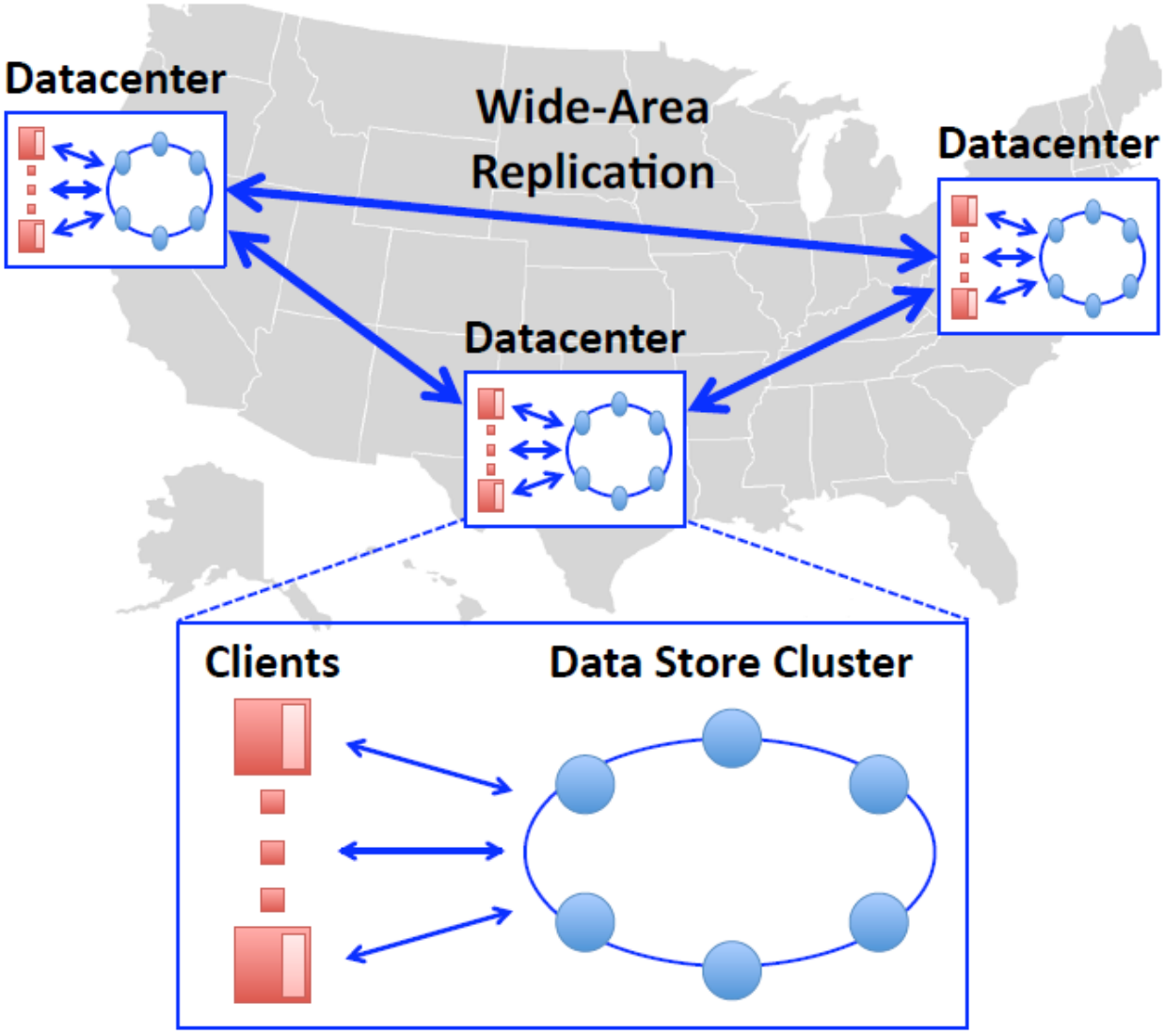
Obiettivi di COPS

Definizione

Un sistema ALPS è un sistema caratterizzato da 4 proprietà:

1. Availability
   Ogni richiesta riceve una risposta su ciò che è riuscito o fallito.
2. Low Latency
   Le operazioni completano velocemente.
3. Partition-tolerance
   Il sistema continua a funzionare nonostante arbitrarie perdite di messaggi: in questi casi alcuni nodi possono rimanere isolati dagli altri e non avere più informazioni aggiornate (sono partizionati).
4. High Scalability
   Possiamo aumentare la capacità del sistema.

Architettura del sistema



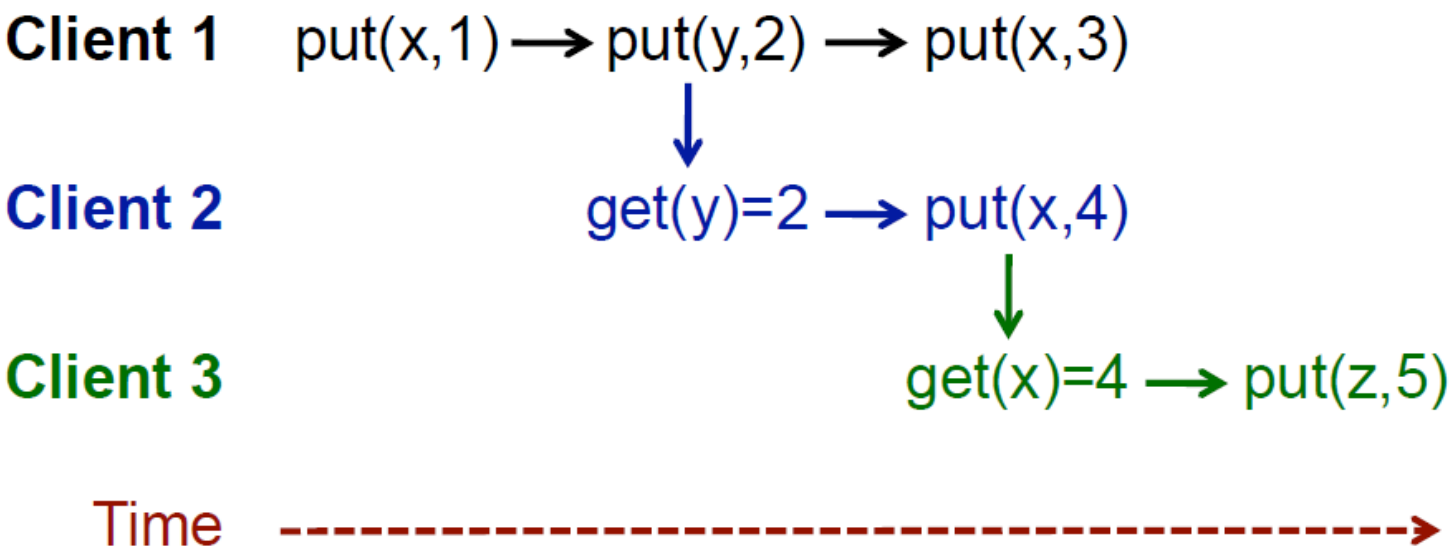
Potenziale di causalità

Definizione

Definiamo il potenziale di causalità fra operazioni (e lo denotiamo con il simbolo $\rightsquigarrow$) come una relazione che soddisfa le seguenti tre regole:

1. **Execution thread.** Se a e b sono due operazioni in un singolo thread di esecuzione allora $a \rightsquigarrow b$ se l'operazione a accade prima dell'operazione b;
2. **Gets From.** Se a è un'operazione di `put` e b è un'operazione di `get` che ritorna il valore scritto da a, allora $a \rightsquigarrow b$;
3. **Transitivity.** Date tre operazioni a, b, c, se $a \rightsquigarrow b$ e $b \rightsquigarrow c$, allora $a \rightsquigarrow c$.

Un esempio


Client 1
put(x,1)
put(y,2)
put(x,3)
Client 2
get(y)=2
put(x,4)
Client 3
get(x)=4
put(z,5)
Time

Casual+ Consistency

Definizione

Definiamo la Causal+ Consistency implementata da COPS come una combinazione di due proprietà:

- Causal Consistency
  I valori restituiti da un'operazione di get da una replica sono consistenti con l'ordine definito da $\rightsquigarrow$;
- Convergent conflict handling
  Tutte le operazioni di put in conflitto vengono gestite nello stesso modo da tutte le repliche, usando una funzione di gestione h (associativa e commutativa).

Definizione

Si dice che due operazioni a e b sono in conflitto se sono entrambe due operazione di put "simultanee" sulla stessa chiave.

Coerenze a confronto

Sequential consistency

Il risultato di qualunque operazione è uguale a quello ottenuto se le operazioni di read e di write da parte di tutti i processi sullo archivio di dati fossero eseguite:

1. in un qualche ordine sequenziale;
2. le operazioni di ogni singolo processo appaiono comunque in questa sequenza nell'ordine specificato dal programma.

Linearizability > **Sequential** > Causal+ > Causal > FIFO > Per-Key Sequential > Eventual

Se in Dynamo abbiamo un livello di coerenza che può oscillare fra Strong ed Eventual, in COPS la definizione risulta più netta: si cerca di implementare il meglio che si può avere con le caratteristiche ALPS!

Astrazioni di COPS (I)

Versione

Ci riferiamo a differenti valori di una data chiave come alla loro versione (e la indichiamo con $\text{key}_{version}$).

Una volta che una data replica in COPS ritorna un certo valore di una data chiave, siamo assicurati, per casual+ consistency, che i successivi valori richiesti a quella replica di quella data chiave saranno quello ottenuto o una sua versione successiva, mai precedente.
(PROPRIETA' DI PROGRESSIONE)

Astrazioni di COPS (II)

Dipendenze

Diciamo che y_i dipende da x_i se e solo se $\texttt{put}(x_i) \rightsquigarrow \texttt{put}(y_i)$.

Tutte le operazioni che causalmente precedono una data operazione devono aver effetto prima che quella operazione avvenga.

Design di COPS (I)

Struttura di COPS

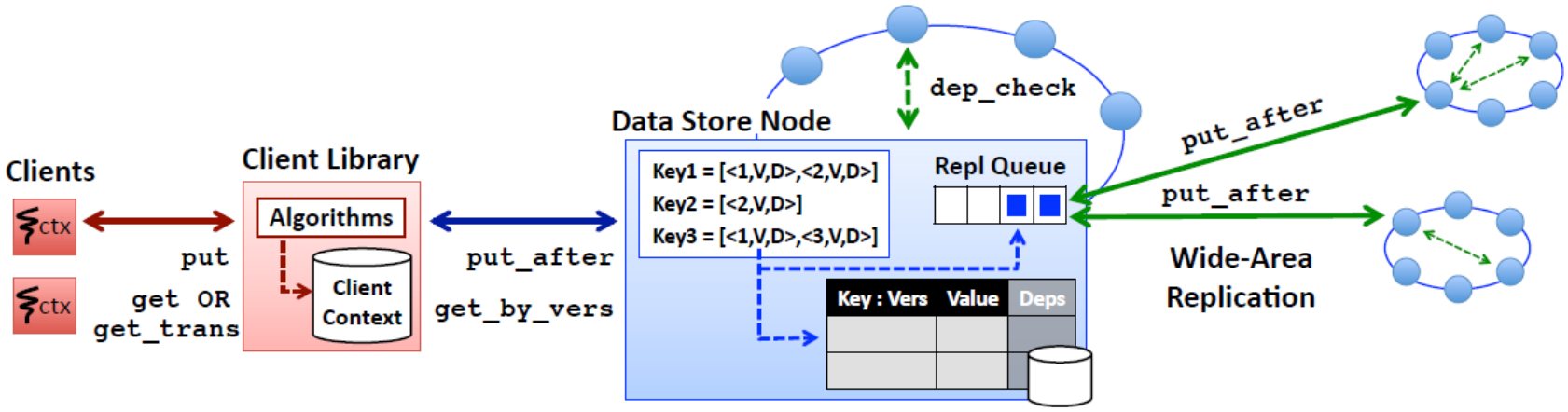
COPS è formato da due componenti principali:

- Key-value store
  Coppie <key, value> alle quali sono associati metadati (numero di versione e lista di dipendenze).
- Client library
  Esporta due operazioni principali alle applicazioni: letture via *get* e scritture via *put*.

Design di COPS (II)

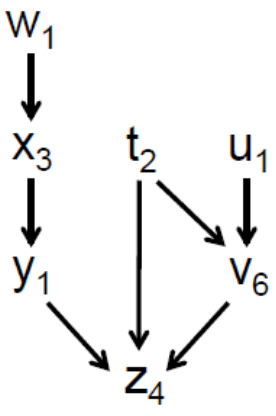
L'interfaccia di COPS è composta da quattro operazioni (più una):

- *ctx_id* ← `createContext()`
- *bool* ← `deleteContext`*(ctx_id)*
- *bool* ← `put`*(key, value, ctx_id)*
- *value* ← `get`*(key, ctx_id)*
- *values* ← `get_trans`*(keys, ctx_id)*



Dipendenze

Il contesto viene usato internamente da COPS per tenere traccia delle dipendenze causali che si vengono a generare quando sono eseguite di volta in volta operazioni di put e di get da parte dei client.
Quella che si viene a creare è quindi una rete di dipendenze come mostrato di seguito:



Val	Nearest Deps	All Deps
t_2	-	-
u_1	-	-
v_6	t_2, u_1	t_2, u_1
w_1	-	-
x_3	w_1	w_1
y_1	x_3	x_3, w_1
z_4	y_1, v_6	$t_2, u_1, v_6, w_1, x_3, y_1$

La proliferazione di metadati riduce l'efficienza: possiamo affidarci alle "nearest dependencies"!

Scritture

Tutte le scritture in COPS:

1. Vengono eseguite sul cluster locale;
2. Vengono propagate in maniera asincrona sui cluster remoti.

Il sistema fornisce due API per gestire questo processo:

- `put_after`
  Consente di eseguire entrambe le operazioni di *write* in un colpo solo e ci assicura che il valore è committato ad ogni cluster solo dopo che tutte le sue dipendenze sono state scritte (su quello specifico cluster).
- `dep_check`
  Verifica che le dipendenze causali siano soddisfatte sul cluster locale, a seguito della ricezione di una `put_after`.

Gestione dei conflitti

Conflict handling

COPS può gestire conflitti in diversi modi a seconda dell'implementazione:

- Last-writer-win-rule
  (detta Thomas Write rule e basata su Timestamp);
- Marking dei conflitti e risoluzione con altri mezzi.

Lamport Timestamp

Vengono usati i Lamport Timestamp per assegnare un numero di versione univoco a ogni aggiornamento. Il nodo setta la parte alta dei bit del numero di versione pari al suo Lamport clock, mentre la parte bassa viene posta pari all'identificativo univoco del nodo.

Non vengono usati i Vector Clock come invece avviene in Amazon Dynamo perché si ritiene che in un sistema troppo grande potrebbero andare fuori controllo.

Get Transaction

Limiti di COPS

Leggere un insieme di chiavi dipendenti usando una singola operazione di get non assicura una coerenza causal+, anche se il data store implementa esso stesso una politica di causal+ consistency.

Novità di COPS-GT

COPS-GT offre un'operazione di `get_trans` che permette di ritornare una visione coerente di più chiavi.

Questo tipo di transazioni non richiedono locks, sono non bloccanti e vengono realizzate al più in due rounds.

Pseudocodice

Presentiamo un esempio di pseudocodice della `get_trans`:

```
# @param  keys    list of keys
# @param  ctx_id  context id
# @return values  list of values

function get_trans(keys, ctx_id):
  # Get keys in parallel (first round)
  for k in keys
    results[k] = get_by_version(k, LATEST)

  # Calculate causally correct versions (ccv)
  for k in keys
    ccv[k] = max(ccv[k], results[k].vers)
    for dep in results[k].deps
      if dep.key in keys
        ccv[dep.key] = max(ccv[dep.key], dep.vers)

  # Get needed ccvs in parallel (second round)
  for k in keys
    if ccv[k] > results[k].vers
      results[k] = get_by_version(k, ccv[k])

  # Update the metadata stored in the context
  update_context(results, ctx_id)

  # Return only the values to the client
  return extract_values(results)
```

Concludendo su COPS

Vantaggi di COPS

COPS è un sistema che offre le seguenti caratteristiche:

- Serve le operazioni localmente, replica in background (always on)
- Partiziona lo spazio delle chiavi in più nodi (scalabilità)
- Controllo delle repliche con il tracciamento delle dipendenze (casual+ consistency)

Svantaggi

- L'uso dei timestamp per la gestione dei conflitti è piuttosto semplice, tuttavia scegliere una modifica a favore di un'altra in questo modo permette potenzialmente di non considerare affatto un eventuale aggiornamento rilevante.

Conclusioni

In un sistema distribuito ci sono sempre due prospettive da tenere in considerazione:

- Il provider
  Vede lo stato interno del sistema: la sua priorità è la sincronizzazione dei processi fra le repliche e l'ordine delle operazioni;
- Un client del sistema di storage
  Il sistema è una black-box: la sua attenzione è rivolta alle garanzie che il sistema distribuito deve fornirgli come parte di un SLA.

Quindi, a seconda delle necessità dei diversi sistemi e della prospettiva client-centrica o data-centrica che scegliamo di adottare, possiamo implementare diversi livelli di coerenza.