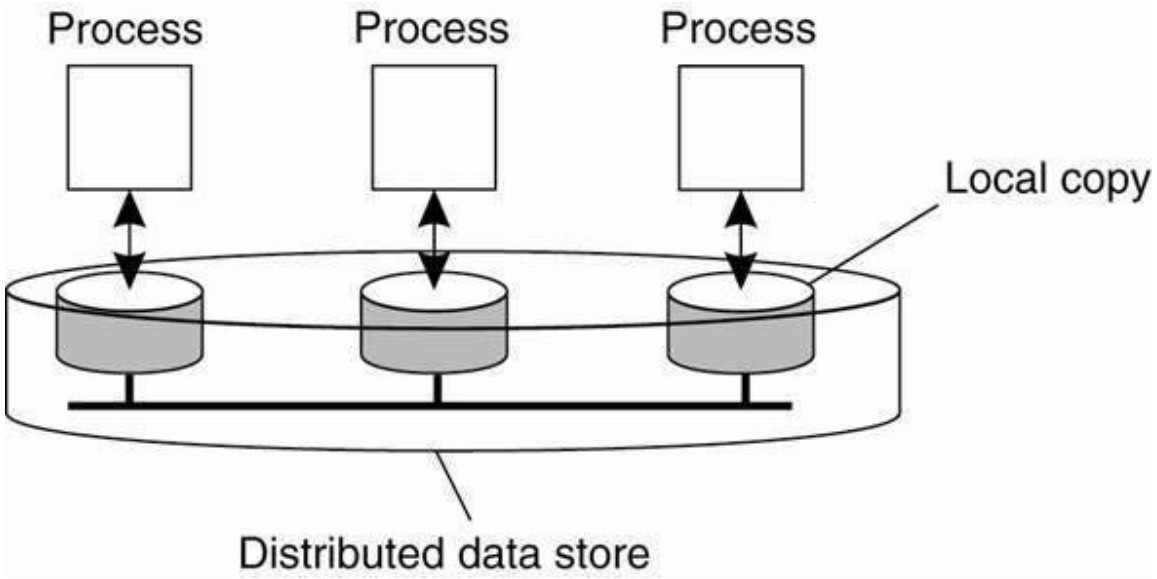
Simple distributed key-value store with consistency guarantees

Alessia Cinelli 0350265

Consistenza nei sistemi distribuiti



- All’interno di un sistema distribuito, è necessario avere dati replicati poiché ne migliora la disponibilità, la tolleranza ai guasti e la scalabilità.
- Una sfida in questi sistemi è mantenere diversi datastore replica coerenti tra di loro.

Livelli di consistenza

Ci sono diverse garanzie di consistenza:

- Alcune più stringenti, consentendo agli utenti di percepire il datastore replicato come se fosse una singola replica. Assicurando una visione uniforme e coerente dei dati tra tutte le repliche.
- Altri livelli di consistenza sono meno stringenti, consentendo un certo grado di discrepanza tra le repliche. Sono solitamente utilizzate per ottimizzare le prestazioni, a discapito di una coerenza assoluta.
- Vedremo in dettaglio la consistenza sequenziale e la consistenza causale.

Consistenza Sequenziale

- La garanzia di consistenza sequenziale impone che tutti i server replica vedono lo stesso interliving di operazioni.
- Interliving ammissibili: sequenza di operazioni che rispettano l'ordine di programma per ciascun processo.
- Per realizzare un datastore distribuito che rispetti le garanzie di consistenza sequenziale si è implementato l'algoritmo di Multicast Totalmente Ordinato.

Consistenza Causale

- Il modello di consistenza causale è un rilassamento della consistenza sequenziale e abbatte l'illusione che ci sia una singola replica all'interno del sistema.
- Per soddisfare la garanzia di consistenza causale, è previsto che tutti i server replica rispettino la relazione di causa-effetto delle richieste dei client.
- Per realizzare un datastore distribuito che rispetti le garanzie di consistenza causale si è implementato l'algoritmo di Multicast Causalmente Ordinato.

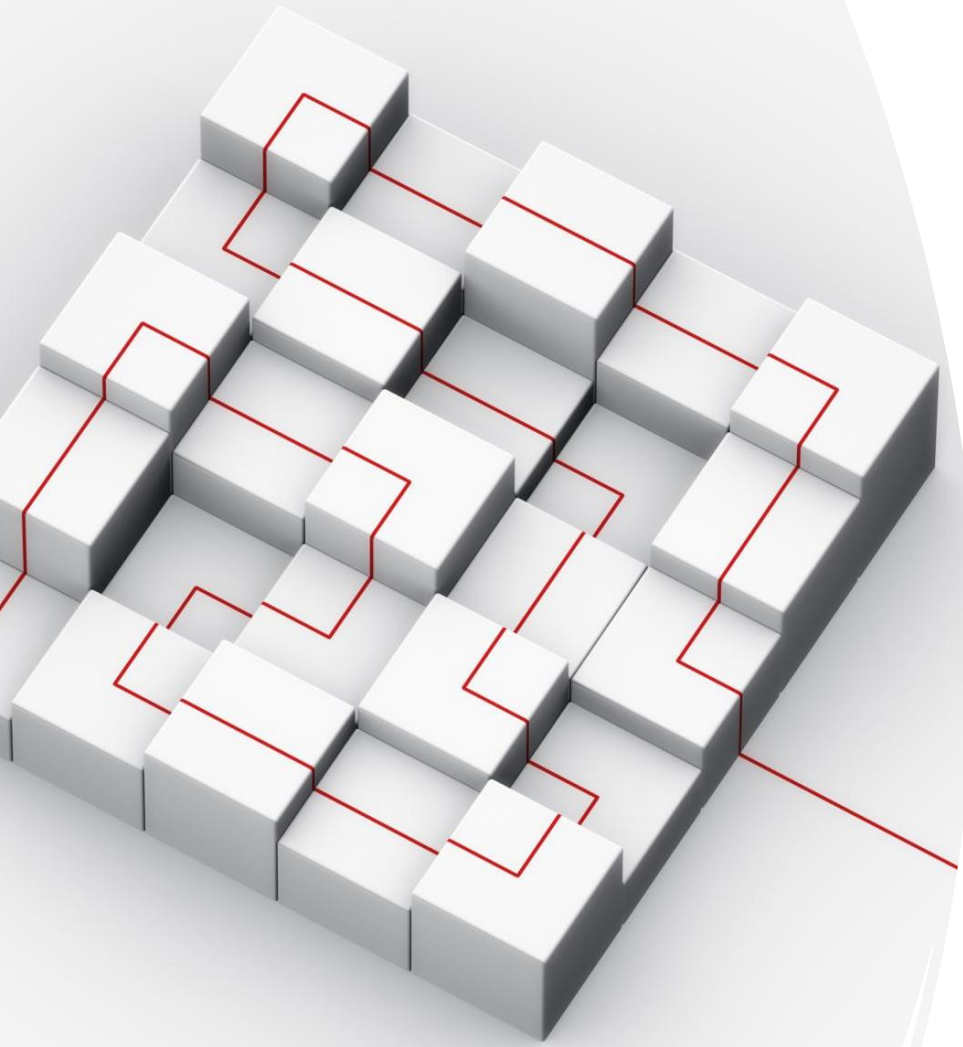
Assunzioni per gli algoritmi di multicast

- Comunicazione affidabile:
- Si è utilizzato il protocollo TCP in relazione con la libreria RPC offerta da GO.

Assunzioni per gli algoritmi di multicast

- Comunicazione FIFO Ordered: i messaggi inviati da un processo ad un altro vengono ricevuti rispettando l’ordine di invio.

- È stato adottato un approccio in cui ogni client contatti sempre una specifica replica del server. In questo modo, ogni richiesta generata dal client viene associata ad un timestamp auto incrementativo, che riflette l’istante di creazione della richiesta. Tale timestamp permette ai server replica di processare le richieste nell'ordine corretto.



Architettura del sistema

- Il sistema è composto da:
- Il Client che può effettuare operazioni di Put, Get e Delete di una key all’interno di un datastore distribuito.
- Uno o più server replica, che offrono garanzie di consistenza di tipo Sequenziale o Causale, a scelta per il Client.

Multicast Totalmente Ordinato

L'algoritmo si comporta in modo differente a seconda che la richiesta effettuata dal client sia un evento interno o esterno.

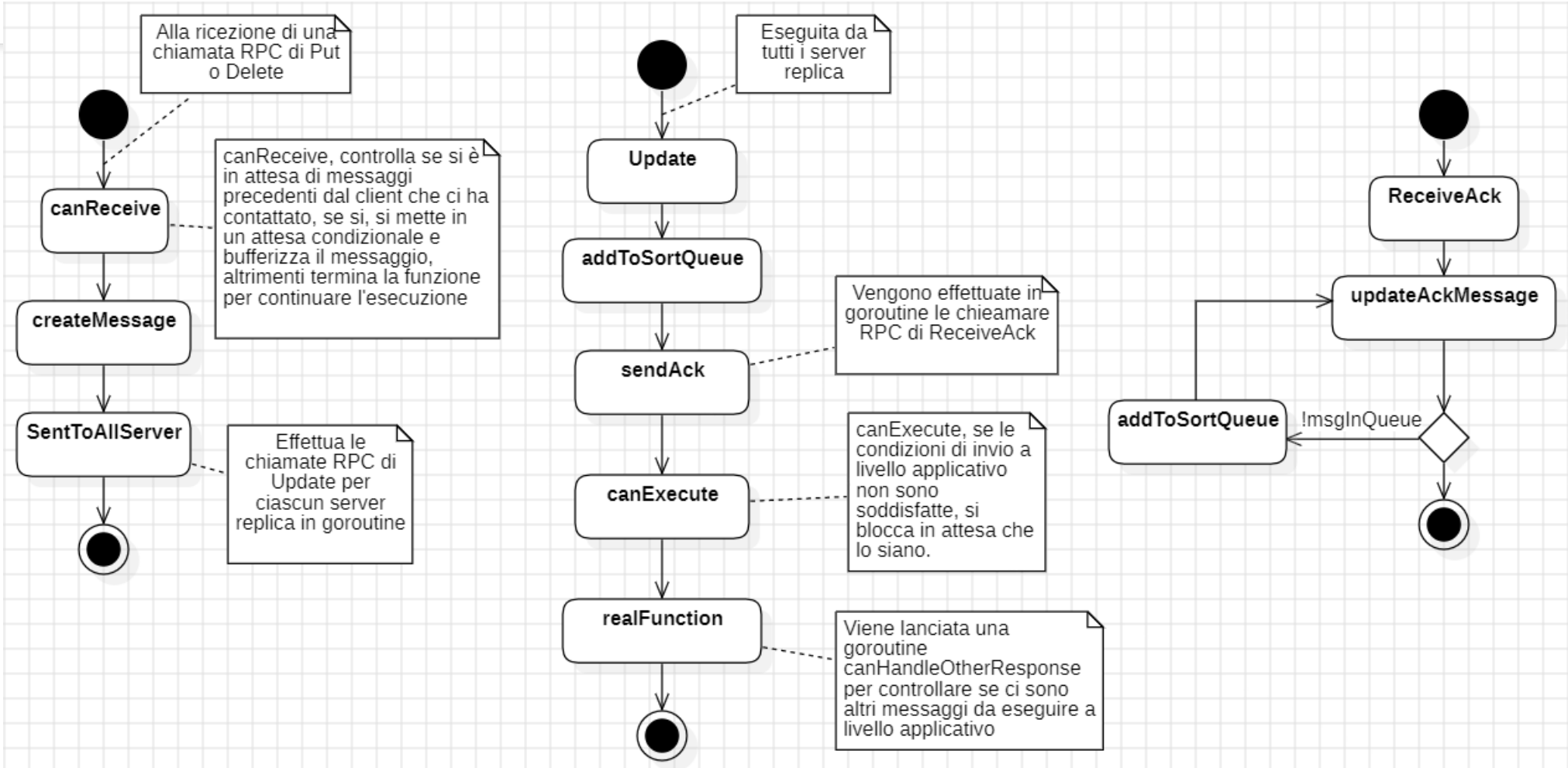
Quando un processo P_i riceve una richiesta da parte del client, gli allega il suo timestamp, generando un messaggio msg_i.

- Se è un evento interno, aggiunge il messaggio alla sua coda locale, ordinata per timestamp.
- Se è un evento esterno invia in multicast agli altri processi (incluso se stesso) il messaggio di update di msg_i.

Ciascun processo ricevente P_j mette msg_i nella coda locale.

- P_j invia in multicast agli altri processi un messaggio di ack della ricezione di msg_i.
- P_j consegna msg_i all'applicazione se:
  1. msg_i è in testa alla coda.
  2. P_j ha ricevuto tutti gli ack relativi a msg_i.
  3. Per ogni processo P_k c'è un messaggio msg_k in coda con timestamp maggiore di quello di msg_i. Ovvero msg_i viene consegnato solo quando P_j sa che nessun altro processo può inviare in multicast un messaggio con timestamp minore o uguale a quello di msg_i.

Realizzazione Algoritmo



Realizzazione Algoritmo – Buffer Message

- Per bufferizzare i messaggi in attesa di essere processati a livello applicativo, ho adottato l'uso di variabili condizionali per la sincronizzazione.
- Quando una richiesta arriva ma non è ancora pronta per essere inoltrata a livello applicativo, il messaggio viene bufferizzato e la goroutine responsabile della gestione della richiesta entra in attesa che un'altra goroutine la sblocchi.
- Il controllo per decidere se sbloccare la goroutine di gestione del messaggio avviene ogni volta che viene ricevuto un ack o quando viene inoltrato un messaggio a livello applicativo.

Realizzazione Algoritmo

```go
func (kvs *KeyValueStoreSequential) controlSendToApplication(message *commonMsg.MessageS) bool {  2 usages  Alessia Cinelli *
    kvs.LockMutexQueue()
    defer kvs.UnlockMutexQueue()

    if kvs.GetServerID() == message.GetSenderID() { // Se il messaggio è stato inviato dal server stesso
        // Non ho ancora inviato tutti i messaggi precedenti chiesti dallo stesso client
        if kvs.GetResponseOrderingFIFO(message.GetClientID()) != message.GetSendingFIFO() { return false }
    }

    if (len(kvs.GetQueue()) > 0 && // Se ci sono elementi in coda
        kvs.GetMsgFromQueue( index: 0).GetIdMessage() == message.GetIdMessage() && // Se il messaggio è in testa alla coda
        kvs.GetMsgFromQueue( index: 0).GetNumberAck() == common.Replicas && // Se ha ricevuto tutti gli ack
        kvs.secondCondition(message)) || // Se per ogni processo pk, c'è un messaggio msg_k in queue
        // con timestamp maggiore del messaggio passato come argomento
        kvs.isAllEndKey() { // Oppure, Se tutti i messaggi rimanenti in coda sono endKey, li elimino

        // Incremento il numero di risposte inviate al determinato client
        if message.GetSenderID() == kvs.GetServerID() {
            kvs.SetResponseOrderingFIFO(message.GetClientID(), ts: 1)
        }

        // Tutte le condizioni sono soddisfatte
        kvs.removeMessageToQueue() // Rimuovo il messaggio dalla coda
        // Aggiornamento del clock del server: Prendo il max timestamp tra il mio
        // e quello allegato al messaggio ricevuto e lo si incrementa di uno
        kvs.updateLogicalClock(message)
        return true

    }
    return false
```

- Condizioni di invio a livello applicativo

```go
// secondCondition ritorna true se:
//   - Per ogni processo pk, c'è un messaggio msg_k in queue con timestamp maggiore del messaggio passato come argomento
func (kvs *KeyValueStoreSequential) secondCondition(message *commonMsg.MessageS) bool {  2 usages  Alessia Cinelli *

    // Controllo che ci sia almeno un messaggio per ciascun server con timestamp maggiore
    // rispetto a quello del messaggio in argomento
    for i := 0; i < common.Replicas; i++ {
        found := false
        for _, msg := range kvs.GetQueue() {
            if msg.GetClock() > message.GetClock() && // il messaggio in coda ha un timestamp maggiore
                msg.GetSenderID() == i { // Se il messaggio è stato inviato dal server i

                found = true
                break
            }
        }
        if !found { return false }
    }
    return true
}
```

Realizzazione Algoritmo

- Le operazioni di Put EndKey:EndValue rappresentano un metodo per far considerare all'algoritmo di Multicast Totalmente Ordinato le ultime operazioni del client, cosi che la seconda condizione dell'algoritmo sia soddisfatta.

- Problematica: numero pari di eventi interni → Possibilità di scelta tramite una variabile d'ambiente

Considerazioni Finali

- Elevato numero di mutex e variabili condizionali sincronizzate.
- Elevati cicli di attesa per il soddisfacimento delle condizioni.
- Elevati numeri di ack scambiati.
- Idea che ci sia una sola replica.

Multicast Causalmente Ordinato

Il seguente algoritmo, al contrario del MTO, non varia la sua implementazione se la richiesta effettuata dal client è un evento interno, o esterno.

Questo algoritmo utilizza i clock logici vettoriali, in cui viene mantenuto un indice per ciascun server replica.

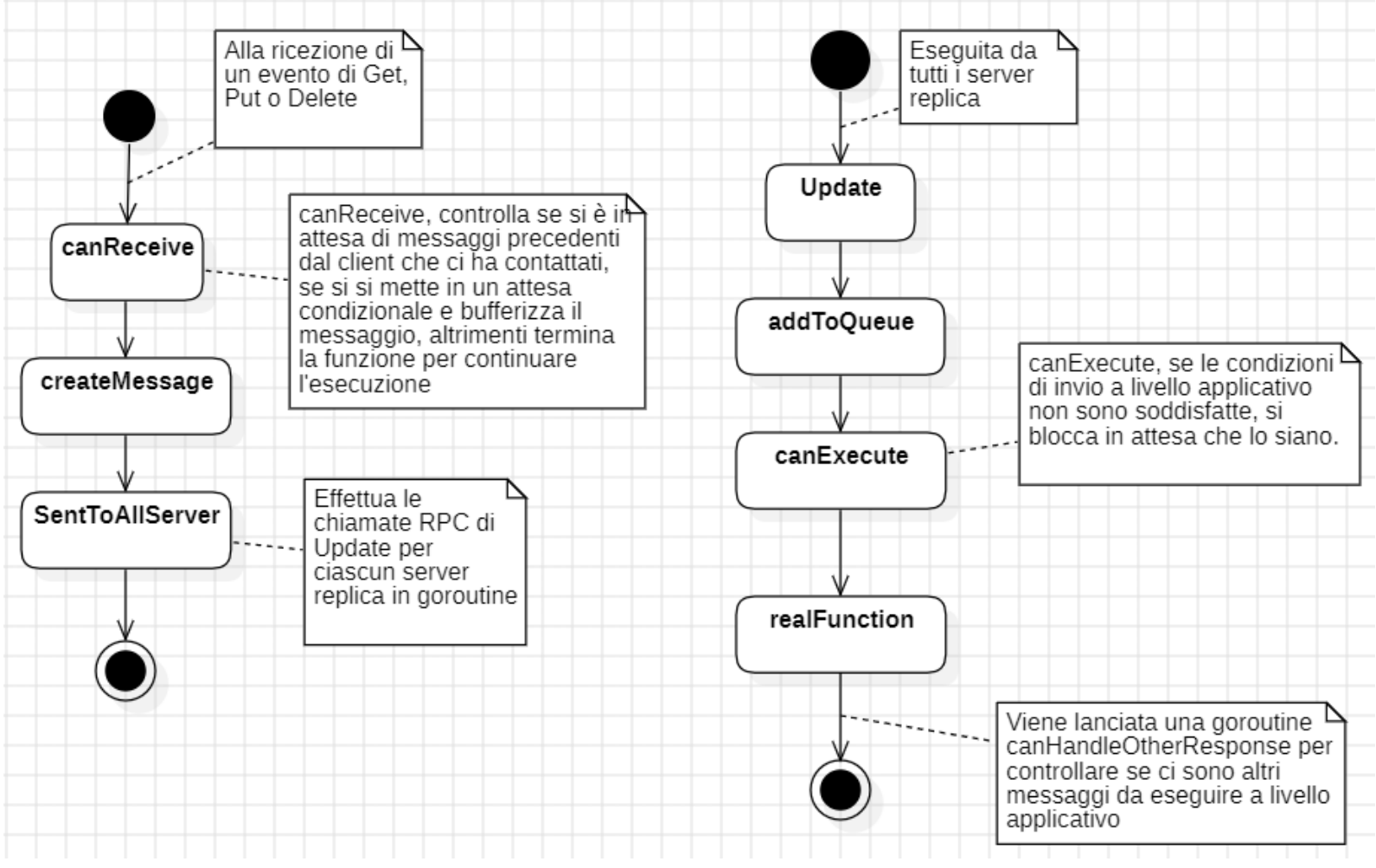
Un processo ricevente Pi di una richiesta da parte del client, gli allega il suo timestamp ts(m) = Vi e invia in multicast agli altri processi (incluso se stesso) il messaggio di update di msg_i.

Ciascun processo Pj incrementa Vj[i] quando riceve un messaggio ad un processo Pi

Un processo Pj che riceve msg da Pi, ne ritarda la consegna all'applicazione (ponendo m in una coda di attesa) finché non si verificano entrambe le condizioni:

1. ts(m)[i] = Vj[i] + 1 m è il messaggio successivo che Pj si aspetta da Pi
2. ts(m)[k] ≤ Vj[k] $\forall\, k \neq i$ ∀ Pk, Pj ha visto almeno gli stessi messaggi visti da Pi

Realizzazione Algoritmo



Realizzazione Algoritmo – Buffer Message

- Similmente al multicast Totalmente Ordinato…
- Per bufferizzare i messaggi in attesa di essere processati a livello applicativo, si è adottato l'uso di variabili condizionali per la sincronizzazione.
- Quando una richiesta arriva ma non è ancora pronta per essere inoltrata a livello applicativo, il messaggio viene bufferizzato e la goroutine responsabile della gestione della richiesta entra in attesa che un'altra goroutine la sblocchi.
- Il controllo per decidere se sbloccare la goroutine di gestione del messaggio avviene ogni volta che viene inoltrato un messaggio a livello applicativo.

Realizzazione Algoritmo

```go
func (kvc *KeyValueStoreCausale) canExecute(message *commonMsg.MessageC) {  2 usages  Alessia Cinelli *
    message.ConfigureSafeBool()

    executeMessage := make(chan bool, 1)
    go func() {
        // Attendo che il canale sia true impostato da canExecute
        message.WaitCondition() // Aspetta che la condizione sia true, verrà impostato in canExecute se è
                                // possibile eseguire il messaggio a livello applicativo
        executeMessage <- true
    }()

    canSend := kvc.controlSendToApplication(message) // Controllo se le due condizioni del M.C.O sono soddisfatte

    if canSend {
        message.SetCondition( b: true) // Imposto la condizione a true
    } else {
        // Bufferizzo il messaggio
        kvc.AddBufferedMessage(*message)
    }
    <-executeMessage // Attendo che la condizione sia true
    // è stata eseguita real function nella goroutine, la risposta è stata popolata.
}
```

- SetCondition e WaitCondition sono utilizzate per bloccare e sbloccare l'esecuzione
- Bufferizzazione del messaggio se non rispetta le condizioni per l'inoltro a livello applicativo

Realizzazione Algoritmo

```go
func (kvc *KeyValueStoreCausale) controlSendToApplication(message *commonMsg.MessageC) bool {  2 usages  Alessia Cinelli
    result := false

    // Verifica se il messaggio m è il successivo che pj si aspetta da pi
    if (message.GetSenderID() != kvc.GetIdServer()) &&
        (message.GetClock()[message.GetSenderID()] == (kvc.GetClock()[message.GetSenderID()] + 1)) {

        result = true

        // Verifica se pj ha visto almeno gli stessi messaggi di pk visti da pi per ogni processo pk diverso da i
        for index := range message.GetClock() { // Per ogni indice del vettore dei clock logici
            if (index != message.GetSenderID()) && // Se l'indice non è quello del mittente del messaggio
                (index != kvc.GetIdServer()) && // e non è quello del server stesso che sta processando il messaggio
                (message.GetClock()[index] > kvc.GetClock()[index]) { // e pj non ha visto almeno gli stessi messaggi di pk visti da pi
                result = false
            }
        }

    } else if message.GetSenderID() == kvc.GetIdServer() { //Ho ricevuto una mia richiesta
        result = true
    }

    // Se è un evento di lettura, controllo se la chiave è presente nel mio datastore
    // Se non c'è aspetterò fin quando non verrà inserita
    if message.GetTypeOfMessage() == common.Get && result {
        _, result = kvc.GetDatastore()[message.GetKey()]
    }

    if result {
        // Entrambe le condizioni soddisfatte, il messaggio può essere consegnato al livello applicativo
        // Aggiorno il mio orologio vettoriale
        if message.GetSenderID() != kvc.GetIdServer() {
            kvc.LockMutexClock()
            kvc.SetVectorClock(message.GetSenderID(), kvc.GetClock()[message.GetSenderID()]+1)
            kvc.UnlockMutexClock()
        } else {
            // Incremento il numero di risposte inviate al determinato client
            kvc.SetResponseOrderingFIFO(message.GetClientID(), ts: 1)
        }

        kvc.removeMessageToQueue(message) // Rimuovo il messaggio dalla coda
        return true
    }
    return false
}
```

Inoltro a livello applicativo

Considerazioni Finali

- Implementazione più semplice rispetto il Multicast Totalmente Ordinato.
- Minor numero di messaggi scambiati.
- Si persa l’illusione di una sola replica.