**Appunti studio Paxos vs Dwork**

**Paxos**

**Introduzione generale**

Si tratta di un algoritmo semplice per la gestione di un sistema distribuito in grado di tollerare errori/problemi. Si tratta di un algoritmo a consenso.

**L’algoritmo a consenso**

Vari processi propongono valori, solo uno viene scelto ed accettato da tutti.

Ci sono 3 ruoli, che possono essere assunti da “agenti”:

1. Proposers: propongono uno (o più) valore/i.
2. Acceptors: accettano uno (o più) valore/i.
3. Learners

Gli agenti possono comunicare tra loro, per decidere (a maggioranza) quale valore accettare.

Requisito 1: un acceptor deve accettare la prima proposta che gli arriva.

Ogni è proposta di valore viene quindi contrassegnata da un numero (univoco) ed un valore.

Requisito 2: se una proposta con un valore viene scelta, allora tutte le proposte accettate con un identificativo maggiore devono avere valore .

Requisito 2a: se una proposta con valore viene scelta, allora tutte le proposte con un identificativo maggiore, accettate da qualsiasi acceptor, devono avere valore .

Requisito 2b: se una proposta con valore viene scelta, allora tutte le proposte con un identificativo maggiore, proposte da qualsiasi proposer, devono avere valore .

Requisito 2c: per ogni ed , se una proposta con valore ed identificativo è presentata, allora c’è un set composto da una maggioranza di acceptors, tale che vale almeno una tra:

1. Nessun acceptor in ha accettato una proposta con identificativo minore di .
2. è il valore della proposta con identificativo più grande tra tutte le proposte con identificativo minore di , prese tra quelle accettate dagli acceptors in .

**Algoritmo base di gestione delle proposte**

Per garantire il rispetto dei requisiti/invarianti stabiliti, l’algoritmo di gestione delle proposte ragiona in questo modo:

1. Un proponente sceglie un nuovo identificativo per la proposta, ed invia una richiesta a tutti gli acceptors in un set scelto, richiedendo in risposta:
   1. Una promessa che non vengano accettate nuove proposte con identificativo minore di .
   2. La proposta accettata con l’identificativo più alto, inferiore ad , se ce ne sono.

Tale richiesta viene chiamata *prepare* request con identificativo .

1. Se il proponente riceve le risposte richiesta da una maggioranza di acceptors, allora può presentare una proposta con un identificativo e valore , dove è il valore della proposta con identificativo maggiore tra le risposte, oppure un valore scelto arbitrariamente dal proponente se nessuna risposta è stata ricevuta.

Il proponente presenta quindi la proposta, inviandola ad un set di acceptor per l’accettazione. Viene chiamata *accept* request.

**Algoritmo finale di gestione delle proposte**

Requisito P1a: un acceptor può accettare una proposta numerata se non ha risposto ad una *prepare* request con un identificativo maggiore di .

Dopo opportune considerazioni sulla gestione delle accettazioni da parte degli acceptors, si ottiene l’algoritmo seguente, composto da 2 fasi:

Fase 1

1. Un proposer seleziona un identificativo ed invia una *prepare* request con numero alla maggioranza degli acceptors disponibili.
2. Se un acceptor riceve una *prepare* request con un numero maggiore di qualsiasi altra *prepare* request a cui abbia già risposto, allora risponde a questa con una promessa, di non accettare alcun’altra proposta numerata meno di , corredata dalla proposta con identificativo maggiore tra quelle accettate (se esiste).

Fase 2

1. Se il proposer riceve una risposta alle sue *prepare* requests (con identificativo ) da una maggioranza di acceptors, allora invia un’*accept* request ad ognuno di quegli acceptors. Tali richieste sono per una proposta con identificativo e valore , dove è il valore della proposta con identificativo maggiore tra le risposte, o un valore arbitrario se nessuna risposta.
2. Se un acceptor riceve un’*accept* request per una proposta numerata , quest’ultimo accetta la proposta, a meno che non abbia già risposto ad una *prepare* request con un identificativo maggiore di .

Essendo prevista la possibilità di ignorare richieste, in caso di situazioni particolari descritte, magari è opportuno informare il mittente di tale scelta.

**Imparare il Valore scelto (Learners)**

Per imparare un Valore scelto, i learner devono venire a sapere che la proposta è stata accettata dalla maggioranza degli acceptors.

Vengono valutare più opzioni per gestire la cosa:

* Tutti gli acceptors avvisano tutti i learner che un valore è stato accettato, ma non è molto efficiente in termini di performance.
* Tutti gli acceptors avvisano un unico learner che un valore è stato accettato e questi poi avvisa il resto dei learner. Questo migliora un po’ le performance, ma il single point of failure è ovviamente un problema.
* L’opzione precedente con un set scelto di learners potrebbe essere la soluzione migliore, e ci si trova quindi in una situazione di bilanciamento tra affidabilità e performance basato sulla cardinalità del set.

Anche con queste valutazioni eventuali fallimenti lato acceptor potrebbero creare problemi e perdita del valore accettato.

**Progressi nello studio**

Per garantire un funzionamento adeguato del sistema, considerando che è quasi sempre possibile porlo in una situazione tale per cui i proposer si mettono in difficoltà tra di loro, è stata pensata una soluzione a singolo proposer eletto.

**Implementazione dell’algoritmo**

Un’implementazione adeguata dovrebbe considerare le seguenti caratteristiche:

* L’algoritmo prevede un network di processi che svolgano tutti e 3 i ruoli precedentemente descritti.
* L’algoritmo sceglie un leader, che svolge il ruolo di proposer eletto e learner eletto.
* Il funzionamento dell’algoritmo è quello descritto in precedenza.
* È necessario uno spazio di archiviazione stabile, per mantenere le informazioni utili agli acceptor anche in caso di fallimento. Un acceptor memorizza la propria risposta nello spazio di archiviazione prima di inviarla.
* È necessario descrivere un meccanismo che garantisca che non vi siano 2 proposte presentate con lo stesso identificativo.

Per l’ultima caratteristica si può considerare che:

* Proposte differenti scelgano il proprio identificativo da set di numeri distinti, così che non possano mai avere lo stesso identificativo.
* Ogni proposta memorizza (nello spazio di archiviazione) il valore della proposta con identificativo più grande che sia stata presentata, ed inizializzano la fase 1 con un identificativo maggiore di quest’ultimo.

**Implementazione di una macchina a stati**

*Vedi nel paper un esempio interessante di implementazione dell’algoritmo in un contesto concreto e reale*.

**Note aggiuntive derivanti da ulteriori fonti**

* L’algoritmo deriva da un esempio storico del Paxos Parliament.
* Vi è poi una simile applicazione, precedente e meno evoluta nel Sinodo cerimoniale dei sacerdoti. Era basato su ballottaggi e gestione di proposte con implicazioni simili a quelle già discusse.
* Dalla discussione sulla gestione del concilio dei sacerdoti, risultano evidenti molte delle osservazioni nate e discusse nel caso tecnico (molto più chiare nel caso tecnico).
* L’algoritmo di Paxon è adatto a sistemi con modesti requisiti di affidabilità, che non giustificano la spesa di un’implementazione real-time estremamente fault-tolerant.
* Con l’algoritmo di Paxon, il tempo non può essere parte dello stato-macchina. Quindi se avviene un fallimento, può volerci un tempo arbitrariamente lungo per eseguire un comando, e il comando di qualcuno potrebbe venire eseguito prima di un comando che era stato inviato precedentemente.
* Il tempo può comunque utilizzare il real-time per la gestione di incarichi esterni a tempo.
* Il protocollo di Paxon è simile ad un protocollo standard con commit a tre fasi. Entrambi, infatti, utilizzano uno scambio di 5 messaggi tra coordinatori e gli altri membri del quorum:
  + Il commit protocol sceglie uno di due valori (commit-abort) mentre il protocollo di Paxon sceglie un decreto in maniera arbitraria.
  + Per convertire un commit protocol in Paxon, qualcuno deve inviare il decreto nel round iniziale di messaggi.
  + Una decisione di *commit* indica che questo decreto è passato, mentre un *abort* indica che un altro è passato.
* La differenza vera e propria dal commit protocol è che il decreto non viene inviato fino alla fase 2. Questo consente al protocollo parlamentare corrispondente di eseguire la prima fase solo una volta per tutti i decreti, quindi è sufficiente lo scambio di soli 3 messaggi per decreto.
* Altri protocolli (come quello di Dwork) sono simili, ma utilizzano ballottaggi sequenziali in round separati.

**Collective consistency – Dwork**

**Base generale**

Per collective consistency si intende un nuovo paradigma per coordinazione di processi multipli all’interno di uno stesso sistema, focalizzato sulla gestione di errori o ritardi nei processi tramite la comunicazione di questi ultimi a tutti i soggetti del sistema.

Per questo sono pensati due elementi importanti:

* Autonomous failure detection
* Consistent reporting protocol

A supporto della Message Passing Interface standard.

**Introduzione - problema**

Prendendo un insieme di computazioni (processi), dividiamo questi in una struttura alternata di fasi di:

* Comunicazione
* Computazione

Le comunicazioni sono gestite tramite la **Message Passing Interface (MPI)** standard, nella quale:

* Ogni partecipante scambia multipli pacchetti data con altri tramite il transport layer.
* Un blocking (waiting) riceve le chiamate presentate dal transport layer per tutti i pacchetti che devono essere ricevuti dagli altri partecipanti.
* Se un partecipante non riesce ad inviare tutti i suoi pacchetti agli altri, tutti gli altri partecipanti vanno in block (wait costante), in attesa di questi pacchetti previsti nella fase di comunicazione successiva.
* Lo stesso partecipante va in block e non può inviare pacchetti, causando problemi a catena.

Quindi, in computazioni reali, il fallimento di un singolo processo blocca l’intero sistema fino ad eventuale interazione umana.

L’obiettivo è quindi quello di sviluppare un metodo in cui tali fallimenti possano essere automaticamente tollerati dai processi rimanenti. La tecnica prevista:

* Consente ai processi rimanenti di accordarsi su una nuova group membership, così che la computazione possa essere riorganizzata nel nuovo gruppo.

Tale accordo è ovviamente impossibile in un sistema pienamente asincrono:

* Un protocollo di agreement che garantisca consistency ha esecuzioni in cui alcuni processi devono bloccarsi o procedere per infiniti step (DDS, FLP).

**Introduzione – parziale soluzione**

Un obiettivo però raggiungibile è la riduzione della finestra (intervallo di tempo) di blocco, durante i quali altri processi vengono bloccati dal fallimento di un singolo:

1. Integrare le chiamate collettive al transport layer con un semplice failure detector [CT, CHT], così che nessun processo blocchi durante la comunicazione collettiva.
2. Eseguire un (preferibilmente semplice) protocollo di collective consistency (CCP), durante il quale alcuni partecipanti potrebbero bloccare, permettendo tali processi di ritornare a condividere una visione consistente di un set di processi non-falliti.

In particolare, tale protocollo, garantisce che 2 processi non si indichino mai a vicenda come falliti e non si blocchino, mantenendo quindi la stessa visione dei membri del gruppo.

1. Riorganizzare la computazione nell’applicazione perché esegua su un nuovo gruppo accettato. Continuando quindi il protocollo di computazione, ignorando messaggi da processi segnalati come falliti.

L’applicazione scelta è particolarmente adatta a questo approccio, data la creazione di un checkpoint naturale dopo ogni comunicazione collettiva (tutti ottengono info utili al backup).

* Però finché i partecipanti hanno visioni differenti del set di processi attivi, la situazione non può migliorare.

**Collective consistency**: viene definita come una debole forma di consenso, nella quale i processi tentano di raggiungere una visione comune del gruppo (“comune” in senso largo).

* Più specificatamente, un processo entra con una visione iniziale, consistente di una lista di processi falliti, e ne esce con una visione in output di processi sospetti di fallimento.

Requisito

La visione in output di ogni processo deve essere in accordo con le visioni in output degli altri processi che supponga non abbiano fallito. Quindi deve essere in accordo solo con processi non in . Quindi se , allora le loro visioni possono essere diverse solo se sospettano l’uno dell’altro.

**Differenze tra collective consistency e consensus**

* La collective consistency consente a 2 processi in ritorno, e , di avere visioni differenti in alcune circostanze.
* Il consensus l’output di può essere diverso da quello di solo se almeno uno di loro non è ritornato.
* La definizione di collective consistency non necessita che tutti i processi non fallimentari terminino.

**Requisiti per collective consistency di failure detection**

**Appunti sulla discussione**

* La definizione è quella riportata precedentemente.
* Viene citata la terza proprietà nelle differenze col consensus.
* Non si parla nello specifico di come i sospetti sui processi vengano generati, fa riferimento ai risultati della Failure detection.
* Un processo che ritorna un output view si dice che ritorna, altrimenti no.

**Collective consistency - debole**

Se è la vista ritornata dal processo e è la vista ritornata dal processo , e se , allora .

**Collective consistency – forte**

Le viste ritornate da tutti i processi sono identiche.

**Quorum system**

Collezione di set di processi, ogni coppia dei quali ha un’intersezione non vuota [Gif, PW, NW]. Possono venire utilizzati per convertire un protocollo a collective consistency debole in uno a forte.

**Monotonicità**

Se è la visione iniziale del processore , e se il processore ritorna , allora .

Che tradotto significa, se sospetta di qualcuno ad un certo punto del processing, allora quel qualcuno resterà sempre sospettato fino alla fine (al massimo può sospettare di altri in aggiunta).

L’esecuzione del protocollo è modellata come una sequenza di eventi, una per ogni processo. Un taglio è l’unione di un set di finiti prefissi di sequenze, una per ogni processo. Il taglio iniziale è vuoto, in quanto avviene prima dell’ottenimento della vista iniziale da parte dei processi.

**Non trivialità**

Il taglio iniziale è multivalente, ovvero, esistono i tagli e contenenti . Tali che in un processo ritorna un’output view vuota e in qualche processo ritorna un’output view non vuota. Data come il set di tutti i processi a parte .

La condizione di non trivialità esclude la soluzione triviale alla collective consistency, nella quale ogni processo ritorna una output view .

Quindi, un protocollo risolve la collective consistency se è non triviale e in tutte le esecuzioni la sua condizione è soddisfatta.

**Obiettivi di design**

* Semplicità: basato sull’assunzione che algoritmi più semplici è più probabile che siano implementati correttamente.
* Velocità: minimalizzazione del numero di passi del protocollo che un processo deve sostenere, prima di inviare l’ultimo messaggio richiesto dal protocollo. Il motivo è legato alla vulnerabilità ai blocchi, durante l’esecuzione del protocollo.

Punto importante per la ricerca

* La failure detection, non è eseguita durante il protocollo. Quindi maggiore è il numero di round di comunicazione, maggiore è la probabilità che un processo si blocchi, tentando di completare il proprio round. Quindi minor numero di round = migliori performance?
* Dipende, in quanto protocollo più complessi possono consentire a più processi di ritornare. Ottenendo quindi più processi disponibili per completare la computazione nell’applicazione.

**Relative ricerche**

*Vengono presentati nel paper vari studi, esperimenti e lavori condotti da vari studiosi sull’algoritmo.*

**Protocollo per collective consistency monotona**

**Descrizione**

* Sono presenti solo fallimenti di tipo crash process.
* I messaggi non sono alterati.
* Ogni messaggio inviato, prima o poi, raggiungerà la sua destinazione, a meno che non avvenga un crash o del mittente o del destinatario.

**Rappresentazione 1**

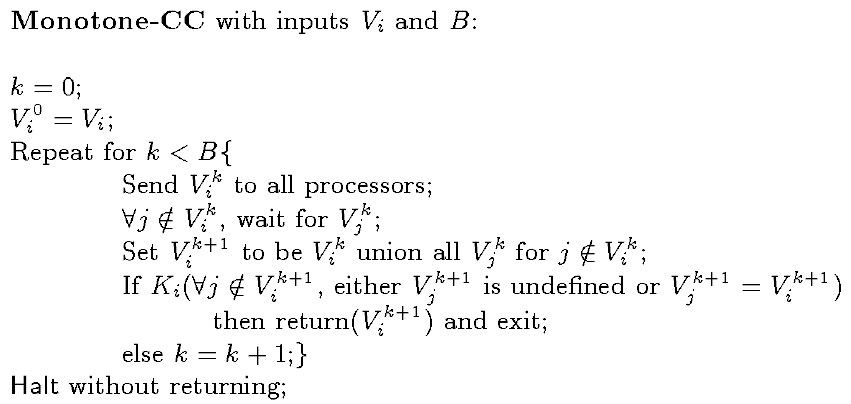
* nella specifica è il knowledge operator, indicando che la veridicità dei suoi argomenti sia conosciuta da processo .
* è usato per denotare il valore di formato durante l’iterazione del processo .
* Se il processo aspetta indefinitamente nell’iterazione , allora abbiamo come .
* Il parametro è il limite superiore del numero di volte che il main loop nel codice viene iterato. Ogni valore di riguarda un protocollo differente.
* Il knowledge operator ed i suoi argomenti, rappresentano un predicato Booleano sconosciuto.

Un’implementazione di tale specifica knowledge based è un algoritmo che sostituisce con uno specifico predicato.

* Un’implementazione sound rimpiazza con un predicato tale che, quando ritorna , allora vale nel set di possibili esecuzioni fornito dal sistema in cui ogni partecipante utilizza al posto di .
* Un’implementazione è detta non triviale se:

1. C’è un’esecuzione in cui ogni input è vuoto e dopo un certo numero di iterazioni per qualche processo ritorna .
2. C’è un esecuzione in cui l’input di qualche processo è non-vuoto e dopo un certo numero di iterazioni per ritorna .

**Algoritmo 1**



**Teorema 1**

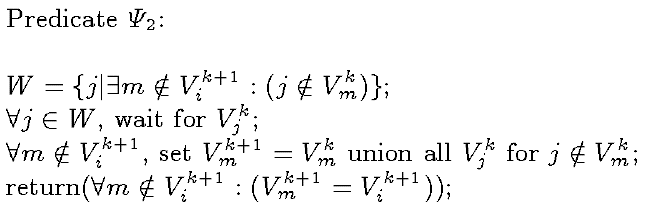
Qualsiasi implementazione sound e non triviale del Procotollo di CC-Monotona, fornisce una soluzione non triviale al problema della collective consistency monotona.

**Rappresentazione 2**

Il nuovo predicato fornisce un’altra famiglia di implementazioni sound non triviali, del protocollo knowledge based precedente.

* Il set contiene i processi , tali che qualcuno di “fidato” secondo possa “garantire” per . Quindi nel caso sopra, questo farebbe in modo che debba aspettare di essere notificato da .
* Utilizzare consente quindi a questo scenario di essere gestito anche con .

**Predicato**



**Discussione 2**

Possiamo quindi trasformare un protocollo per collective consistency debole in uno per collective consistency forte, tramite l’aggiunta di un requisito:

* Il complementare di un set in output deve essere un membro di uno precedentemente scelto dal quorum system.

Possiamo anche usare il quorum system per superare una debolezza dei protocolli che soddisfano la specifica knowledge-based della versione 1:

* Tutti tali protocolli bloccano se un qualche processo fallisce all’inizio del protocollo, perché ogni altro processo aspetta di ricevere la vista dei processi falliti.

**Rappresentazione 3**

Dato un quorum system:

Sostituendo 2 righe nella prima rappresentazione e definendo , si può fornire un’implementazione sound e non triviale per la specifica risultante.

Fornendo quindi una soluzione per il problema della collective consistency forte e monotona.

**Un limite inferiore per protocolli di collective consistency**

**Introduzione**

Non esiste un protocollo di collective consistency che sia in grado di garantire la terminazione non-bloccante di alcun partecipante in presenza di un processo lento o fallito.

Infatti, per ogni protocollo di collective consistency vale una tra:

* Esiste un processo tale che in qualche esecuzione ogni processo deve aspettare di essere notificato da .
* C’è un’esecuzione in cui nessun processo fallisce, ma nessun processo fornisce mai un output.

**Definizioni e assiomi**

Vedi definizioni di monotonicità e non trivialità già descritte, per un’idea dell’environment.

Un protocollo è non triviale se il taglio vuoto è multivalente.

Un taglio è univalente se esiste un taglio contenente tale che in qualche processo ha input dato, ma non è multivalente.

* Si dice che un taglio univalente ha projected value 0 se è contenuto in un taglio, in cui il set vuoto è dato come output.
* Altrimenti si dice che ha projected value 1.

Un taglio è nullavalente, se non è contenuto in alcun taglio con un evento di output.

Un processo decide , se è multivalente e per tutti i tagli che contengono , tali che agisca in , è univalente.

Un taglio è detto non deciso, se è multivalente o nullavalente.

Un processo è pronto ad un taglio , se esiste un evento in e un taglio , tale che .

Un processo è permanentemente bloccato a , se non c’è alcun taglio che contiene , tale che agisca in . Si noti che se è permanentemente bloccato ad un taglio multivalente , allora trivialmente decide .

Questi risultati si applicano a tutti i sistemi che rispettino i seguenti assiomi:

* **Compatibilità di Azioni Indipendenti**: dato un taglio, e dati tagli contenenti tali che e , dove è un singolo evento che accade al processo (vale anche per e ). L’assioma di compatibilità dice che se , allora è un taglio.
* **Separabilità di Eventi**: dati e eventi distinti, contenuti in un taglio . L’assioma di separabilità afferma che c’è un taglio contenente uno di questi eventi, ma non l’altro.

**Il limite inferiore**

Lemma 2

Per tutti i tagli , al massimo un processo può decidere ed essere pronto a .

Lemma 3

Se è multivalente e non decide , allora esiste un taglio contenente tale che agisce in e è non deciso.

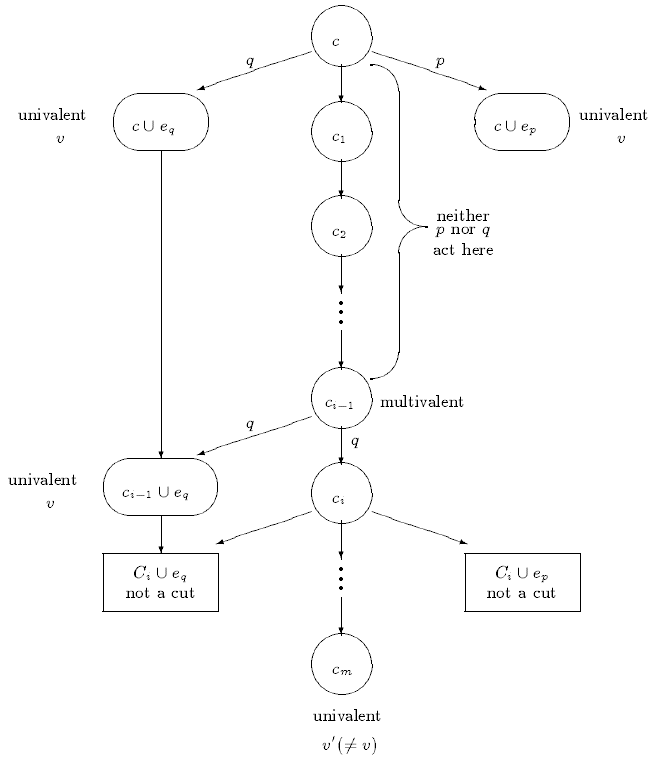
Lemma 4

Se è nullavalente e non è permanentemente bloccato a , allora esiste un taglio contenente tale che agisce in e è non deciso.

Lemma 5

Se il taglio è non deciso, allora c’è un taglio non deciso contenente tale che, per tutti i processi vale una tra:

* decide .
* è permanentemente bloccato a .
* agisce in .



**Teorema**

Per ogni protocollo non triviale , esiste un set contenente al massimo un processo tale che, per tutti gli , esiste un taglio non deciso tale che per tutti i , o non è pronto a , oppure in almeno eventi sono accaduti a . In aggiunta, questi tagli non decisi formano una catena di contenimento.

**Differenze riscontrate tra i 2 lavori**

* Paxon si concentra su un’attiva comunicazione tra i soggetti del sistema (per evitare errori), mentre Dwork dà per assunto che gli errori avvengano e si concentra sull’avvisare tutti che questi errori sono avvenuti (e dove magari).
* Dwork dà inoltre grande importanza alle visioni che un processo ha del sistema, ovvero a chi è attivo e fa bene il suo lavoro, in contrapposizione a chi ha causato errori e verrà quindi ignorato. Quindi qui la consistency si trova tra le visioni dei vari processi.

*Da capire meglio il discorso dei tagli alla fine di Dwork.*

**Valutazione dei linguaggi C# vs Ada vs GO**

**C#**

**Aspetti positivi**

* Linguaggio orientato agli oggetti C-style, molto simile a quelli già utilizzati, quindi rapido da imparare. Soprattutto per me (Alessandro) che ho avuto occasione di utilizzarlo moltissimo in stage.
* Garbage Collector 😊
* Più adatto in generale per lo sviluppo (suite di test, debugging(?), profiling, memory management).

**Aspetti negativi**

* Non nativamente pensato per distribuzione e concorrenza. Durante lo stage io (Alessandro) ho dovuto implementare package specifici per integrare la concorrenza senza generare eccezioni.
* C# implica essere legati allo sviluppo di applicazioni in .NET (aiuto).

**Ada**

**Aspetti positivi**

* Utilizzato e proposto da Tullio, sono quindi presenti numerose proposte e descrizioni da lui fornite per il corso.
* Estremamente orientato alla distribuzione, con package esplicitamente utili al progetto che noi dobbiamo realizzare:
  + Memoria condivisa accessibile da tutti i processi (buona per Paxos).
  + Gestione nativa di Nodi, task e chiamate a metodi remoti.
* Linguaggio molto preciso, in cui le best practices vengono introdotte direttamente nel compilatore (aka. Generalmente se una cosa va bene a lui allora dovrebbe andare bene in generale).
* Esistono varie guide semplici, fatte da fanatici del linguaggio. È facile da imparare partendo da basi di linguaggi C.

**Aspetti negativi**

* Non conosciuto (forse qualcosa Nicola?), quindi va preventivato un piccolo periodo di test ed apprendimento del linguaggio (1 settimana per iniziare?).
* Considerato un linguaggio praticamente morto, *quindi eventuali implementazioni sarebbero comunque da considerarsi unicamente a scopo dimostrativo del problema. Poco utili ad un effettivo utilizzo reale*.
* Niente Garbage Collector ☹

**GO**

**Aspetti positivi**

* Garbage Collector 😊
* Possiede un meccanismo di gestione della concorrenza, il che lo rende ottimo per la codifica di programmi per macchine multicore e network.
* Linguaggio abbastanza versatile, nonostante sia staticamente tipizzato, il che lo rende una sorta di versione più moderna di Ada.
* Buone performance, ambiente deterministico (niente randomicità a meno di Seed).
* Documentazione autogenerabile (figo)
* Possibile creare binary eseguibili, senza necessità di compilatore.

**Aspetti negativi**

* Non conosciuto, quindi va preventivato un piccolo periodo di test ed apprendimento del linguaggio (1 settimana per iniziare?). Tullio ne era poco entusiasta.
* Documentazione del linguaggio si dice faccia un po’ schifo (d’altro canto, quale documentazione non fa schifo?)
* Si dice che la gestione delle dipendenze sia un po’ fastidiosa.
* Gestione non classica dei thread, utilizzando delle sorta di “channel”, abbastanza brutto.
* Sono presenti solo strutture di tipo Maps, questo è importante da chiarire, perché gli algoritmi che dobbiamo implementare (in particolare Dwork) fanno uso di questa struttura.
* La gestione delle variabili è molto simile a javaScript, *quasi peggio oserei dire* (AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAHHHHHHHHH).
* La gestione della concorrenza viene fatta in maniera molto rudimentale (mutua esclusione con lock manuali e funzioni da creare a DoC), *in questo senso era persino meglio C# che almeno ti dava la libreria già fatta ed autogestita.*

**Alcune particolari differenze tra GO e Ada**

* L’ambiente di GO è deterministico (niente random a meno di Seed), mentre quello di Ada no.
* Le procedure/funzioni di Ada non possono ritornare Tuple (più di un valore), in GO sì.
* In GO le variabili hanno sempre un valore di default (in base al tipo), *potrebbe non essere una cosa così buona, se vogliamo valori nulli*.

**Punti di studio degli algoritmi**

**Dworks**

1. Studio legato al tradeoff tra il rendere il protocollo sofisticato (più round di comunicazione) e vulnerabilità ai fallimenti (più round più fallimenti), in particolare legato ad algoritmi non necessariamente di collective consistency.

*Magari Paxos non viene considerato esplicitamente di collective consistency e potrebbe essere valutato.*

1. Esplicito riferimento nel paper all’MPI come implementazione, con eventuali revisioni, e un’esplorazione della possibilità di communication package con failure detection tramite timeout. (vedi paper aggiuntivo sull’MPI)
2. Definire chiaramente (a livello pratico) il significato di Quorum System in questo contesto.

**Paxos**

1. Ricercare un’esplicita definizione di “collective consistency”, generalmente si parla di “consensus algorithm” nei paper.
2. Studi di vari ricercatori dicono che un algoritmo affidabile per eleggere un proposer dovrebbe utilizzare randomicità o real time, in particolare si parla di timeout (anche nel punto 2 di Dworks si parla di timeout! Anche se in un contesto di failure detection).

**Generale**

1. Entrambi gli algoritmi parlano di gestione di processi, quindi un PoC potrebbe essere utile per esplorare gli elementi in comune per impostare un environment che ne esalti le differenze.
2. Disambiguare il soggetto attore principale nei 2 algoritmi: nodi e processi. E come andrebbero rappresentati poi all’interno dei vari linguaggi.
3. GO e Ada non supportano nativamente la creazione di GUI, quindi eventuali programmi di test dovranno essere console-based. (Probabilmente non un grosso problema)

**Elaborazione dei punti di studio**

1. Buono spunto come base per gli esperimenti da condurre sul PoC.
2. Da definire a seguito di una lettura del paper in esempio, ed eventuali altre implementazioni…
3. **Quorum**: sottoinsieme di un insieme totale di nodi.

**Quorum System:** collezione di quorum, con la proprietà che ogni coppia di quorum ha un’intersezione non vuota.

1. Paxos di fatto non esprime una definizione esplicita di Collective Consistency, però sappiamo che si tratta di un **consensus algorithm**! Questo è importante perché effettivamente Dworks ci dà già delle differenze esplicite tra Collective Consistency e Consensus:

* In consensus l’output di può essere diverso da quello di solo se almeno uno di loro non è ritornato.
* La definizione di collective consistency non necessita che tutti i processi non fallimentari terminino.

Queste differenze sembrano effettivamente coincidere con il comportamento dell’algoritmo di Paxos:

* In Paxos le richieste che potrebbero portare a fallimenti o inconsistenze non vengono nemmeno gestite dagli Acceptors, quindi vengono ignorate e non ritornano.
* In Paxos le richieste gestite ed accettate dovrebbero tutte portare ad una terminazione positiva del processo, si parla infatti di “valori accettati persi” nel caso di malfunzionamenti (situazione gestita tramite solid storage).

**Proposte di implementazione**

* Pseudo-blockchain, dove problemi matematici vengono dati in pasto ad un sistema di nodi, che dovranno provare a trovare la soluzione corretta ed approvata dal sistema stesso.