Relazione Progetto Unico
Programmazione Concorrente e Distribuita

Emanuele Lamagna

Marzo 2023

Indice

Capitolo 1

Programmazione concorrente

1.1 Analisi preliminare

La prima parte del progetto richiede di scrivere un programma che, data un certa directory e una certa parola, cerchi tutti i pdf presenti nella directory e nelle sottodirectory e conti tutte le occorrenze di tale parola. Questo deve avvenire in modo concorrente, utilizzando cinque diversi approcci:

- a thread
- a task
- ad eventi
- reattivo
- ad attori

Oltre a ciò dev'essere presente una GUI che mostri in tempo reale l'andamento della ricerca, ossia visualizzando il numero di file analizzati, il numero di pdf trovati e il numero di occorrenze di tale parola. La GUI deve infine avere vari pulsanti (start, pause, stop), un campo per la scrittura della parola scelta e un pulsante per inserire il path su cui effettuare la ricerca.

In seguito verranno illustrati i dettagli relativi ai vari approcci.

1.2 Strategia risolutiva e architettura proposta

L'approccio scelto per la soluzione del problema è un'architettura **Master-Worker**. Questo tipo di architettura si presta ottimamente al problema

poiché permette di suddividere i due compiti principali richiesti, ossia la ricerca dei file e la ricerca della parola nel file. Infatti l'idea è di utilizzare un Master Thread per scorrere tutti i file e trovare, tra di essi, quelli che sono in formato .pdf: una volta fatto ciò, se il file soddisfa questa condizione, viene interpellato un Worker Thread, che si occuperà di analizzare tale pdf per cercare almeno un'occorrenza della parola scelta. Questo tipo di progettazione, inoltre, si rispecchia molto nel problema dei produttori e consumatori: infatti il Master Thread si comporta da produttore, in quanto "produce" i percorsi dei pdf da analizzare e li mette su un buffer, mentre i Worker Threads si comportano da consumatori in quanto "consumano" il percorso per cercare la parola inserita.

A seguire i vari approcci con i quali è stata realizzata l'architettura richiesta. Tutti sono stati realizzati in Java.

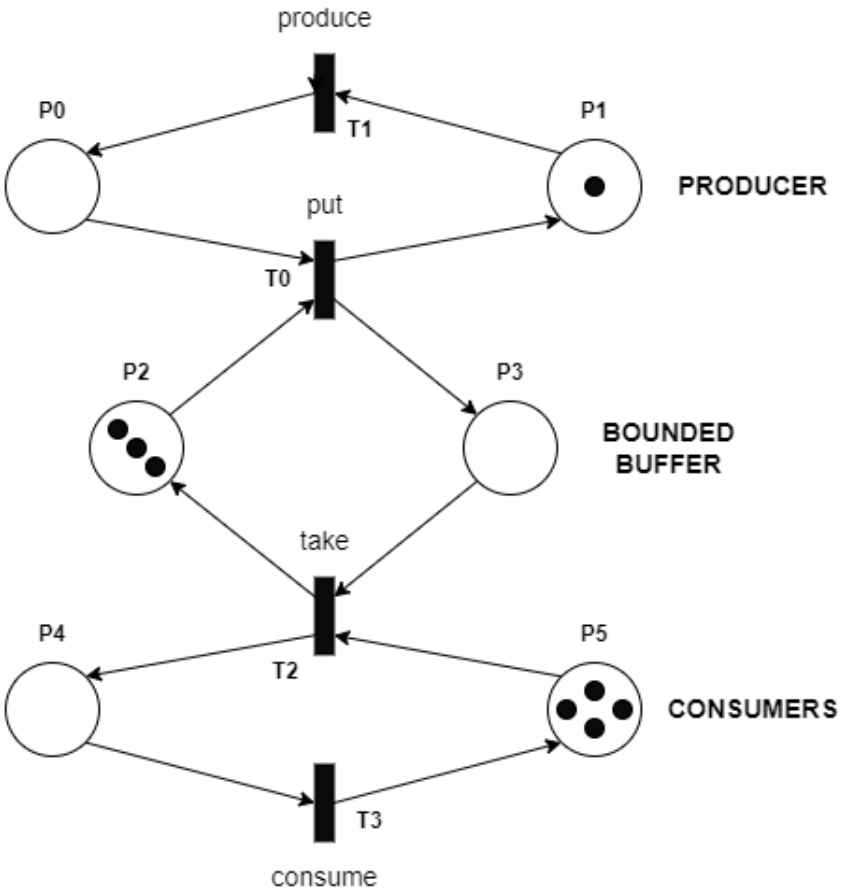
1.2.1 Approccio a thread

Per l'approccio a thread sono stati utilizzati i **Monitor**: questo è un meccanismo che permette di gestire la mutua esclusione (problema inevitabile in un caso di questo tipo) utilizzando metodi identificati dalla keyword *synchronized*: questa permette di evitare, in automatico, l'accesso concorrente al blocco di codice in questione. Oltre a ciò sono state sfruttate le funzionalità offerte da *wait*, *notify* e *notifyAll* che, rispettivamente:

- mette in pausa il thread corrente;
- sveglia uno dei thread in pausa;
- sveglia tutti i thread in pausa.

Oltre al Master thread vengono quindi, all'avvio, creati *n + 1* Worker threads, con *n = n. processori*, per sfruttare al meglio i core del pc: il thread in più è utile perché, anche in casi di utilizzo intenso, i thread possono andare in pausa o in page fault per qualche motivo particolare non ben specificato. In questo caso quindi il thread in più è utile per far viaggiare la CPU al massimo delle sue prestazioni.
Per la gestione dei pdf è stata utilizzata una *BlockingQueue*: essa, tramite i metodi *take* e *put*, permette di rimuovere/aggiungere alla coda. La particolarità di questi due metodi, però, consiste nel fatto che, se all'esecuzione l'operazione non è possibile (quindi un *take* su coda vuota o un *put* su coda piena) il thread va in pausa, fino a quando l'operazione non sarà possibile. Questo meccanismo permette di evitare al meglio il busy waiting.

Rete di Petri

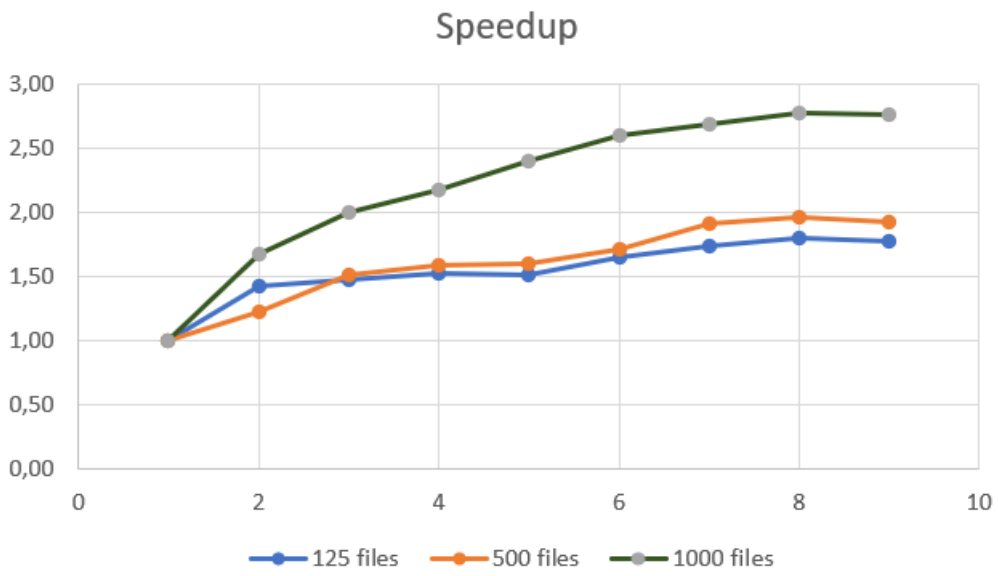


Prove di performance

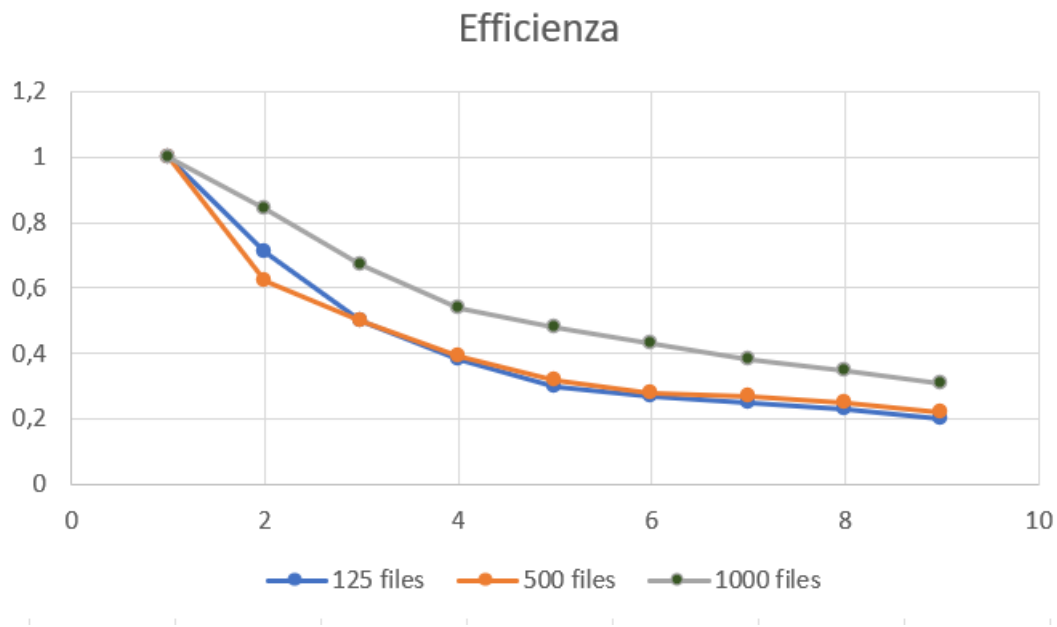
Ho provato ad eseguire il programma cambiando un po' i parametri per vedere cosa sarebbe successo. Ho scelto di effettuare le prove con 125, 500 e 1000 file, e ogni opzione svolta con un numero di thread da 1 a 9. La seguente tabella riporta i tempi di esecuzione in millisecondi:

		Files		
		125	500	1000
Threads	1	14593	46539	127369
	2	10283	37853	75622
	3	9885	30885	63476
	4	9560	29654	58734
	5	9677	29162	53176
	6	8838	27272	49043
	7	8403	24454	47326
	8	8126	23757	45890
	9	8242	24248	46122

Possiamo notare che in generale, aumentando i thread, le performance tendono a migliorare. Il miglioramento più ampio è sempre quello che si ha passando a 2 thread, poi fino a 4 c'è un miglioramento lieve. Dal 5 fino ai 9 thread c'è un lievissimo miglioramento complessivo, ma i risultati sono un po' altalenanti. Sicuramente possiamo aspettarci un qualcosa di simile: i test sono stati effettuati su una macchina con otto core, quindi ha senso che all'avvicinarsi degli 8/9 thread le performance aumentino. C'è da considerare, però, che per programmi con molte operazioni di I/O è consigliabile utilizzare più thread. Nel nostro caso, infatti, abbiamo molte letture da file. Applichiamo altri due strumenti utili alla misura delle performance. Il primo è lo speedup:



Con questo grafico i dati diventano molto più esplicativi. Si vede subito, infatti, che da 125 files a 500 files non abbiamo un significativo speedup, mentre il passaggio a 1000 files porta uno speedup notevole. Probabilmente sarebbe stato più consono effettuare l'analisi con insiemi di file di grandezza più ampia (es. 200, 1000, 10000) per avere risultati più consoni, perché presumibilmente aumentando di molto i file (es. 10000) avremo uno speedup maggiore. L'ultimo strumento che possiamo utilizzare è l'efficienza:



Possiamo notare due cose: la prima è che, all'aumentare dei thread, l'efficienza tende a diminuire (complice la loro creazione e i meccanismi di sincronizzazione, più thread abbiamo e più dobbiamo gestire queste dinamiche). La seconda è che, come avevamo già intuito, all'aumentare dei file, l'efficienza tende anch'essa ad aumentare.

Verifica tramite JPF

Per essere sicuro di non ottenere casi di deadlock o livelock ho svolto delle prove con JPF (Java Path Finder), uno strumento che analizza tutti i possibili percorsi generati dal non determinismo d'esecuzione dei thread concorrenti. Siccome i percorsi sono moltissimi, ho deciso di svolgere la verifica con 2 thread e un numero di file limitato. Il risultato dell'esecuzione è il seguente:

```
====================================================== results
no errors detected

====================================================== statistics
elapsed time:       00:02:23
states:             new=423827, visited=1047579, backtracked=1451752, end=2
search:             maxDepth=591, constraints=0
choice generators:  thread=433172 (signal=9383, lock=12273, sharedRef=395459, threadApi=210, reschedule=24536), data=0
heap:               new=58503, released=46548, maxLive=454, gcCycles=1300278
instructions:       11838700
max memory:         368MB
loaded code:        classes=95, methods=2164

====================================================== search finished: 11/24/22 11:58 PM
```

Tra i numerosi parametri e dati specificati all'interno del risultato, possiamo evincere che il programma (nelle sue condizioni di verifica) è esente da errori quali deadlock o livelock.

1.2.2 Approccio a task

Per l'approccio a task sono stati utilizzati gli Executor di Java. Con gli Executor è possibile creare dei task che andranno poi ad essere "sottomessi", e che restituiscono delle future. L'idea, quindi, è stata di creare un task-tipo (il WordSearchTask) che estende Callable<Integer>: in questo modo la Future restituita è una Future<Integer>. In che modo quindi viene utilizzata? Questa Future restituisce 0 se il pdf non contiene la parola ricercata, 1 se la restituisce. Da fuori, ogni volta che una future restituisce un risultato, viene aggiornato il contatore dei file che contengono la parola: in questo modo un risultato positivo farà crescere il contatore di 1, un risultato negativo lo lascerà invariato. Il meccanismo pause-resume, come anche nel caso dell'approccio reattivo, è stato implementato con un piccolo busy-waiting: infatti è stato posto un while su una condizione all'inizio di ogni ricerca da parte del Master, che va in sleep e si sveglia ogni 50 millisecondi. La prima soluzione era la stessa dell'approccio a thread, quindi con i Monitor: in un secondo momento ho pensato che quello fosse un meccanismo che non c'entrasse nulla con gli Executor, e quindi ho optato per una soluzione di questo tipo. Un approccio più legato agli Executor sarebbe potuto essere relativo al metodo shutdown che permette ai task dell'Executor di terminare, ma elimina quelli in coda. Il problema, in questo caso, sarebbe stato trovare il modo di far ripartire l'esecuzione dei task rimasti. Il busy waiting non è mai una scelta eccellente, però in questo caso credo che sia una scelta abbastanza giustificata.

1.2.3 Approccio ad eventi

Per l'approccio ad eventi è stato utilizzato il framework Vert.x, che permette di utilizzare un event loop per la gestione degli eventi. L'idea di base è

semplice: creare un Verticle e utilizzare il meccanismo dell'event loop per adempiere ai nostri scopi. A livello implementativo è stata creata una classe, AnalyzerAgent, che estende AbstractVerticle: in questo modo la classe diventa di fatto un Verticle. La comunicazione avviene attraverso l'EventBus: tramite esso è possibile scambiare messaggi in vari canali per poi eseguire determinate azioni. I due meccanismi utilizzati sono i metodi send (per inviare messaggi su un certo canale specificato) e consume (al contrario, per consumare messaggio dal canale specificato). In seguito una lista dei canali utilizzati e i loro scopi:

- pause: la view manda un messaggio su questo canale per notificare il model della pausa
- resume: la view manda un messaggio su questo canale per notificare il model della ripresa
- next: è il canale grazie al quale l'analisi va avanti, infatti ogni messaggio corrisponde all'obbligo di analizzare il prossimo file fornito dallo stream
- analyzed: canale che serve a capire quando incrementare la variabile che conta i file analizzati
- matching: canale che serve a capire quando incrementare la variabile che conta i file che contengono la parola scelta
- found: canale che serve a capire quando incrementare la variabile che conta i file che sono PDF
- mastefinished: canale che serve a capire quando è finito il flusso principale

1.2.4 Approccio reattivo

Per l'approccio reattivo sono stati utilizzati due stream: uno principale, che controlla i file e notifica quando trova i PDF, e uno secondario che, a partire dal PDF, cerca la parola in questione. Questi due stream, che sono Flowable di RxJava, sono hot stream, ossia emettono di continuo anche se non ci sono subscribers, e non iniziano un nuovo stream per i nuovi subscribers. Il Flowable principale emette quindi stringhe, ossia i nomi dei file che sono stati identificati come PDF, mentre quello secondario emette valori booleani, che indicano se la parola è presente nel file oppure no. La backpressure strategy scelta per questi stream è quella basata su un buffer. Ma chi, di fatto, si sottoscrive ai Floawable? La risposta è la GUI, in maniera diretta,

passando però dal Controller: infatti, alla pressione del tasto START, viene effettuata la sottoscrizione agli stream, specificando la funzione da eseguire alla ricezione dei dati (ossia l'incremento delle varie variabili e il conseguente aggiornamento grafico).
In modo identico all'approccio a task, anche qui il meccanismo pause/restart è gestito con un piccolo busy waiting, con un loop su una variabile di condizione che effettua il controllo ogni 50 millisecondi. Anche in questo caso ci sarebbero state, in maniera molto probabile, soluzioni migliori, ma tutto ciò avrebbe portato il progetto ad un livello di difficoltà superiore a quello richiesto.

1.2.5 Approccio ad attori

Per l'approccio ad attori è stato utilizzato Akka, in linguaggio Java. In questo approccio tutto è gestito da attori, in ordine:

- AnalyzerActor: l'attore che si occupa di analizzare i file e trovare i PDF
- SearcherActor: l'attore che si occupa di cercare da parola in questione nei file che riceve dai messaggi dell'AnalyzerActor
- CounterActor: l'attore che si occupa di gestire tutti i contatori (file totali, trovati e matching), e anche di gestire il meccanismo di fine della ricerca. Inoltre comunica ogni cambiamento al ViewerActor, che poi provvede ad aggiornare la GUI.
- ViewerActor: l'attore che riceve informazioni sui cambi dei contatori e sullo stato della ricerca e che, di conseguenza, provvede ad aggiornare la GUI.

I messaggi scambiati tra gli attori si raggruppano in due gruppi, ossia i protocolli:

- CounterProtocol: racchiude i messaggi che servono ad incrementare i tre contatori, più quello che serve a chiudere la ricerca
- SearchAnalyzeProtocol: include il messaggio di boot, i messaggi per il pause/resume, per la search, per l'aggiornamento e il reset della GUI

1.2.6 Differenze tra i vari approcci

Dopo aver finito di implementare tutti gli approcci, si può giungere a qualche considerazione conclusiva. L'approccio a thread è quello più veloce fra

tutti: questo è giustificato dal fatto che gli altri approcci sono più di alto livello, mentre qui, ad esempio, andiamo a gestire manualmente aspetti di sincronizzazione. Al contrario, l'approccio ad attori è il più lento: in maniera opposta a alla considerazione precedente, si può dire che il meccanismo degli attori è ad un livello molto alto, e permette di astrarre molti aspetti della logica, quindi la "lentezza" di esecuzione è giustificata. Il vantaggio è che, creando semplicemente creando attori e protocolli, riusciamo a gestire tutte le comunicazioni, una cosa non di poco conto.
L'approccio ad eventi (gestito in questo caso con Vert.x) e quello reattivo (Rx) sono in qualche modo simili: entrambi sfruttano canali per inviare comunicazioni e, sulla base di queste ultime, effettuano operazioni consone. Non sono però la stessa cosa: Vert.x si basa su event loop racchiusi nei Verticle e, sfruttando l'Event Bus, invia messaggi su certi canali specificati, mentre Rx si ispira al pattern observer e sfrutta gli hot/cold stream e il maccanismo di sottoscrizione ad essi per fornire le informazioni.

Capitolo 2

Programmazione distribuita

2.1 Analisi preliminare

La seconda parte del progetto richiedeva di modificare un programma fornito in modo tale da renderlo distribuito. Il programma di partenza, scritto in Java, ha come obiettivo la risoluzione di un puzzle: infatti viene fornita un'immagine divisa in caselle, ma ogni casella è posizionata in ordine casuale, rendendo il puzzle completamente mischiato. Selezionando due caselle, esse si spostano. L'obiettivo è, attraverso questi scambi di caselle, riformare l'immagine originale.
Lo scopo del progetto è quello di rendere il puzzle "giocabile" da più giocatori (il limite è stato fissato a 4), i quali possono interagire con esso in

qualsiasi momento. Inoltre, ogni giocatore deve vedere il puzzle allo stesso modo di tutti gli altri, quindi i cambiamenti devono essere real-time e subito verificabili da ognuno (quindi, in ognuna delle GUI).

2.2 Strategia risolutiva e architettura proposta

Innanzitutto, c'è da puntualizzare che il linguaggio scelto per la realizzazione del programma è Scala. Questo, ovviamente, implica che è stato necessario convertire e adattare il codice di partenza da Java a Scala (fortunatamente abbastanza indolore, per ovvi motivi).

```
akka {
    actor {
        provider = "cluster"
        warn-about-java-serialization-usage = false
        allow-java-serialization = true
    }
    remote {
        artery {
            enabled = on
            transport = "tcp"
            canonical.hostname = "127.0.0.1"
        }
    }
    cluster {
        log-info = off
        seed-nodes = [
            "akka://ClusterSystem@127.0.0.1:2551",
            "akka://ClusterSystem@127.0.0.1:2552",
        ]
    }
}
```

La soluzione è stata realizzata grazie ad Akka Cluster, un'estensione di Akka utile proprio a scrivere programmi ad attori distribuiti: infatti permette di realizzare programmi P2P decentralizzati, senza il rischio di incappare in point of failure o colli di bottiglia, sfruttando Akka Remoting Artery (sopra, in foto, il file di configurazione). Quest'ultimo utilizza TCP o Aeron

UDP come mezzi di trasporto affidabili, ed esibisce la proprietà di location transparency (ossia l'assenza di differenza, a livello API, tra sistemi locali e sistemi remoti). Per quanto riguarda Akka in sé, invece, è stato utilizzato in versione classica, a differenza della prima parte del progetto dov'era stato utilizzato in versione tipizzata.

A livello implementativo ogni giocatore è rappresentato da un attore, il PuzzleActor: esso effettua l'override di alcuni metodi forniti dalla classe Actor, ossia preStart (dove si iscrive al cluster), postStop (dove si disiscrive dal cluster) e receive (dove descrive il comportamento da adottare a seconda dei messaggi in arrivo). Per quanto riguarda il receive, è necessaria una disamina più approfondita degli handler relativi ai vari messaggi. Innanzitutto abbiamo MemberUp e MemberRemoved, di ClusterEvent, che si verificano quando, rispettivamente, un attore entra nel cluster e quando viene rimosso. I restanti principali sono definiti dall'object PuzzleDomain:

- EnterRoom. Quando un nuovo attore entra nel cluster, gli altri membri vengono notificati. Alla ricezione del messaggio, gli attori che già erano nel cluster inviano il puzzle allo stato attuale al nuovo entrato, che potrà poi svolgere le operazioni esattamente come gli altri. Il puzzle viene inviato come insieme di Tile , che ovviamente devono essere serializzabili.

- TileSelected. Quando viene selezionata una Tile in un certo nodo (quindi un certo giocatore seleziona una casella), viene chiamato un metodo (sendSelection) che permette di mandare in broadcast a tutti gli altri nodi il messaggio di selezione della Tile. Questi, quindi, mostreranno la Tile selezionata. Il meccanismo di SelectionManager, poi, capirà se scambiare le caselle (se premiamo, in successione, due Tile diverse) o se, semplicemente, disattivare la Tile (se premiamo, in successione, la stessa Tile).

- PuzzleMessage. Qui un attore riceve il puzzle allo stato attuale (come insieme di Tile) e imposta il suo puzzle interno uguale a quello ricevuto. Questa operazione gli permetterà di "sincronizzare" il proprio puzzle con quello di tutti gli altri.

Possiamo dire che l'approccio utilizzato sia molto soddisfacente, poiché il sistema si comporta come da specifiche: all'entrata, il giocatore riceve una copia dell'attuale puzzle; agli spostamenti ad opera di un qualsiasi giocatore, tutti gli altri visualizzano immediatamente l'aggiornamento grafico; la terminazione da parte di un giocatore non influisce sugli altri; non ci sono coordinatori o server centrali; gli utenti giocano in modo asincrono.

2.2.1 Gestione della sezione critica

Quando due attori vogliono modificare il puzzle (ad esempio, selezionare una tile quando nessuna è selezionata, oppure scambiare tile quando una è già selezionata) potrebbe accadere un conflitto. Infatti, prendendo come riferimento un puzzle con 3 righe e 5 colonne, se ad esempio la tile in (0,1) è selezionata, potrebbe succedere che un attore scelga di premere la tile (0,0), mentre un altro attore, simultaneamente, di premere la tile (1,2): ovviamente i due attori hanno intenzioni differenti, ossia provare due soluzioni diverse per la risoluzione del puzzle. In linea generale, per la risoluzione della sezione critica in maniera non centralizzata, si può usare l'algoritmo di Ricart-Agrawala: seguento questo algoritmo il processo che vuole accedere alla sezione critica manda un messaggio a tutti gli altri processi per informarli e ricevere l'ok da parte di tutti: se tutti danno l'ok significa che può entrare, mentre, al contrario, deve mettersi in coda per eseguire la sua azione appena la risorsa da utilizzare è libera. Nel nostro caso, però, la coda non ci serve: infatti, se prendiamo il caso analizzato prima, se l'attore che arriva prima è quello che vuole scambiare la tile in (0,1) con quella in (0,0), allora lui svolgerà la sua azione, però a questo punto lo scambio che voleva fare il secondo attore non ha più senso. Infatti ora le carte in tavola sono cambiate, e la situazione che ha spinto l'utente a fare quella determinata scelta non esiste più. Quindi la soluzione scelta è la seguente:

- un attore che vuole interagire con il puzzle manda un messaggio a tutti gli altri attori
- gli altri attori rispondono in maniera positiva se non sono interessati ad entrare in sezione critica, o se hanno richiesto l'accesso dopo il primo attore (viene controllato tramite dei timestamp), oppure rispondono in maniera negativa se sono dentro la sezione critica
- se l'attore che ha iniziato la richiesta riceve tutti ok, allora entra in sezione critica ed effettua l'azione, altrimenti non fa nulla: quest'ultimo caso sta a significare che un altro attore ha svolto una mossa, e la sua è stata annullata

In sostanza l'algoritmo scelto è una versione modificata di quello di Ricart-Agrawala, che si adatta al nostro caso.