Relazione sul progetto WATOR – Sistemi Operativi e Laboratorio

0.0| Il progetto WATOR

Il progetto *WATOR* si propone di ricreare un modello di simulazione che possa emulare il comportamento di alcuni esseri viventi e l'interazione tra essi. Il pianeta su cui questi esseri compiono il proprio ciclo di vita è appunto chiamato **Wator** ed è interamente ricoperto di acqua. Tutti gli esseri che dunque lo abitano sono di tipo acquatico e nello specifico si tratta di squali e pesci; predatori in continua ricerca di cibo che condividono il proprio spazio con prede in continuo movimento caotico.

La superficie del pianeta è rappresentata attraverso una matrice le cui celle sono occupate da uno squalo, da un pesce o semplicemente da acqua.
Ogni essere adotta un comportamento specifico, cercando di sopravvivere nel tempo e di riprodursi quanto più possibile.
Nello specifico:

- Ogni squalo, se ne ha lo spazio, si riproduce e successivamente si muove cercando di divorare un pesce. Se non sono presenti prede nelle celle limitrofe, lo squalo effettua un movimento casuale. In aggiunta se lo squalo non mangia per un numero prestabilito di unità di tempo, muore di fame.
- Ogni pesce, se ne ha lo spazio, si riproduce e poi si muove in maniera casuale.

I movimenti effettuati all'intero del pianeta sono di una singola cella per unità di tempo per ogni essere vivente. La struttura del pianeta è sferica, per cui ci si aspetta che movimenti che superano un'estremità della matrice, portino ad occupare celle dell'altro lato.
L'unità di tempo all'interno della simulazione viene detto chronon e scandisce ogni aggiornamento.

1.0| Infrastruttura del progetto

Il progetto si basa sull'interazione di due processi che comunicano tra loro: uno chiamato *wator* (cuore pulsante della logica applicativa) e l'altro *visualizer* (componente adibito alla visualizzazione dell'elaborato e all'interazione con ogni tipo di output).
Il primo di questi immediatamente permette la nascita di *visualizer* come suo sottoprocesso.
I due si scambiano messaggi attraverso un semplice protocollo di comunicazione ideato per ridurre al minimo possibile il numero di byte trasmessi sul canale. I dettagli del protocollo verranno discussi in seguito.

Il processo *visualizer* ha il compito di aspettare che *wator* elabori correttamente i nuovi dati della matrice e che, dunque, sia pronto ad inviarglieli. Il processo *wator* invece non ha necessità di attendere, potendo continuare la propria esecuzione in maniera totalmente autonoma.
Proprio per la natura di questa infrastruttura di comunicazione, il processo *wator* si presta perfettamente ad impersonare il ruolo di *client*, mentre il processo *visualizer* ha tutte le caratteristiche per ricoprire la parte del *server*.
Il canale sfruttato per lo scambio di informazioni tra i due processi protagonisti è una socket sulla quale entrambi scrivono e leggono dati.
La tipologia di socket impiegata all'interno del progetto è quella *AF_UNIX* che concretizza

al meglio il canale di comunicazione locale, utilizzando come path specifico per il file: ***/tmp/visual.sck***, che verrà creato a run-time dal processo *visualizer* ad ogni esecuzione.

Il processo *wator* adotta una struttura operativa a *farm*, che permette in modo efficiente di modificarne il contenuto velocizzando quanto più possibile l'esecuzione attraverso l'impiego di molteplici thread; ognuno dei quali adibito a lavorare solo su un'area assegnatagli.
Il processo *visualizer*, invece, opererà adottando una struttura iterativa semplice, capace dunque di consumare una sola connessione per volta.

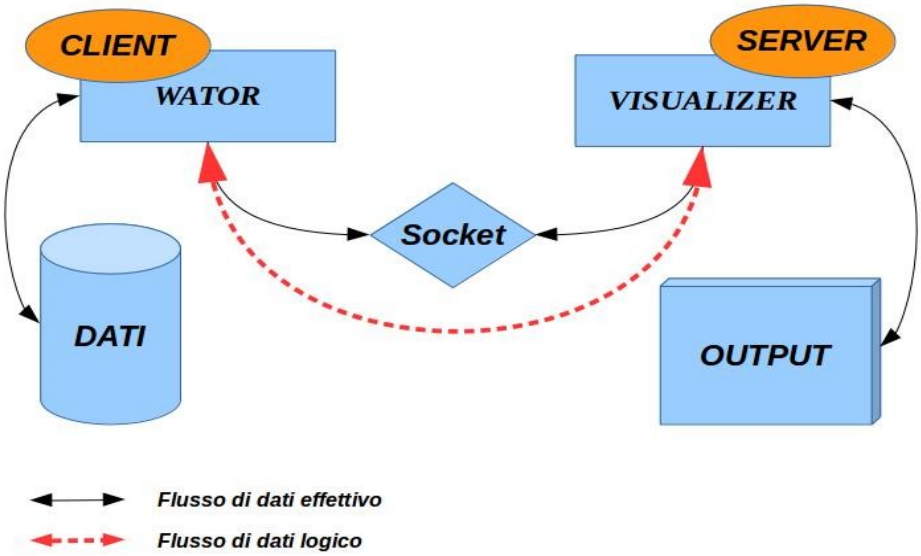

Illustrazione 1: Schema dell'infrastruttura generale

1.1.0| Il processo wator e la struttura a farm

Come accennato in precedenza, il processo *wator* opera attraverso un'architettura a farm: costituita dunque da un thread *dispatcher*, un thread *collector* ed *N* thread *worker*. Il numero *N* viene stabilito a run-time attraverso opzioni da riga di comando o utilizzando una costante di default.

Ogni *worker* dovrà occuparsi dell'aggiornamento dei valori contenuti all'interno di una propria sottomatrice; ci saranno dunque molteplici entità operanti sulla stessa risorsa e questo ci pone davanti al problema di dover assicurare l'integrità delle strutture dati, che dovranno rimanere consistenti durante l'intera esecuzione.
Il compito di individuare tutte le sottomatrici N x K (definite attraverso opportune macro) da assegnare ai “lavoratori” appartiene al thread *dispatcher.*
Il *collector*, invece, dovrà assicurarsi che ogni *worker* completi l'aggiornamento della propria area di competenza e dovrà, di conseguenza, operare solo quando ogni altro thread avrà terminato la propria esecuzione. Sarà sempre il *collector*, in aggiunta, a doversi interfacciare con il processo *visualizer* e a dover, di conseguenza, gestire la comunicazione attraverso la socket.

Ho riflettuto a lungo a quale struttura potesse assicurare una certa sincronia tra i vari thread dell'architettura, e questo continuo “dover attendere la terminazione dell'altro” alla fine mi ha portato a pensare che i rispettivi protagonisti del processo di elaborazione fossero come dei server autonomi e dei client allo stesso tempo: ognuno di essi fornisce un servizio in risposta alla richiesta da parte dell'altro e successivamente richiede un servizio. Questo processo di comunicazione ciclico necessitava di una struttura che permettesse loro di cumulare un “pool” di messaggi da parte del thread precedente nella catena di e di consumarli tutti prima che il thread successivo cominciasse la propria esecuzione.
La struttura che mi sembrava più idonea per adempiere a questo compito è quella della coda: ogni client utilizza la coda del server successivo come “deposito” del proprio elaborato e si mette in attesa che qualcuno aggiunga dati sulla propria coda, affinché possa ricominciare l'elaborazione. Ogni elemento della struttura a *farm* è dunque associato ad una propria coda dei messaggi.
L'unico problema da dover fronteggiare una volta scelta la struttura da implementare per gestire la comunicazione fra i thread era l'approccio da utilizzare per creare sincronia tra di essi. Sembrava necessaria la presenza di una *condition variable* per ogni coda, che permettesse al thread “lettore” di non fare attesa attiva aspettando che il precedente finisse il proprio lavoro, e di una *mutex* che permettesse la scrittura e la lettura di informazioni in mutua esclusione.
Si è scelto, dunque, di creare una struttura *synqueue* rappresentante appunto una coda di informazioni che permette di sospendersi durante la lettura in caso la coda fosse vuota, e permette di aggiungere elementi gestendo in maniera opportuna la *sezione critica*.

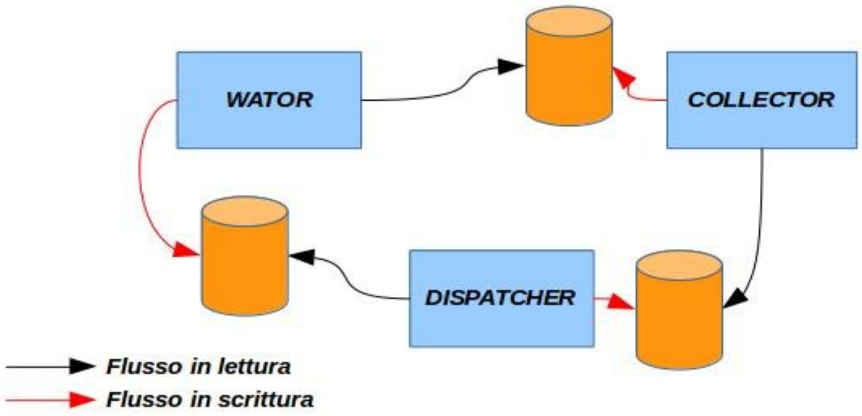

Illustrazione 2: Architettura di comunicazione del processo wator

Wator si occupa di gestire due tipi di segnali: il segnale *SIGINT* (che viene inviato in fase di terminazione del programma) e il segnale *SIGUSR1* (che serve a forzare il programma ad effettuare il dump attuale della matrice). La problematica maggiore da affrontare era quella di sviluppare una gestione che preservasse l'integrità della matrice, e che dunque tenesse conto della terminazione del lavoro svolto dai *worker*.
In particolare, a seguito di un *SIGINT* il programma affettua un'ultima stampa della matrice e poi termina e, a seguito di un *SIGUSR1*, effettua un dump valido a preceduto da un aggiornamento completo e non parziale delle celle.

Per assicurare che tutto ciò fosse garantito, si è lasciato che il *dispatcher* e i worker gestissero l'arrivo di uno dei segnali impostando precisi flag e che a reagire di conseguenza fosse unicamente il *collector*: l'unico in grado di avere unicamente i dati aggiornati e completi.
E' dunque sempre il *collector* l'unico che può decidere di far crollare l'infrastruttura e di causare la terminazione di tutti i thread e successivamente dei due processi. Quando questo accade, il *collector* effettua l'inserimento di un elemento *NULL* sulla coda del *dispatcher*, che a sua volta effettua un inserimento dello stesso valore per ogni thread *worker* sulla loro coda. Dopo ogni inserimento di *NULL*, il thread mittente termina la propria esecuzione.
La chiusura del processo *visualizer* avviene attraverso messaggi specifici inviati sul *socket.*

1.1.1| I thread worker

L'accesso da parte dei *worker* alla matrice del pianeta dev'essere sincronizzato per evitare che due o più thread differenti possano elaborare contemporaneamente le informazioni di una stessa cella, dunque si individua un'altra *sezione critica*. Ogni essere può muoversi sempre in *4* differenti direzioni, delineando in questo modo una “croce” di possibili celle limitrofe occupabili. Si presenta la necessità di poter utilizzare delle *lock* che possa assicurare la mutua esclusione. Ho scelto di utilizzare una matrice di *mutex* avente le stesse dimensioni della matrice originale del pianeta, così che ogni thread possa effettuare una *lock* della *mutex* in posizione *(i, j)* per poter accedere in maniera sicura alla posizione *(i, j)* della matrice del pianeta.
Bisogna però effettuare una *lock* anche delle celle limitrofe rispetto a quella presa in considerazione, poiché a causa uno spostamento (o della riproduzione) potrebbero essere accedute.
L'ordine con cui le *lock* vengono effettuate è determinante per evitare situazioni di *deadlock* fra i vari *worker.* Quello utilizzato da me effettua le lock da sinistra verso destra, dall'altro verso il basso.

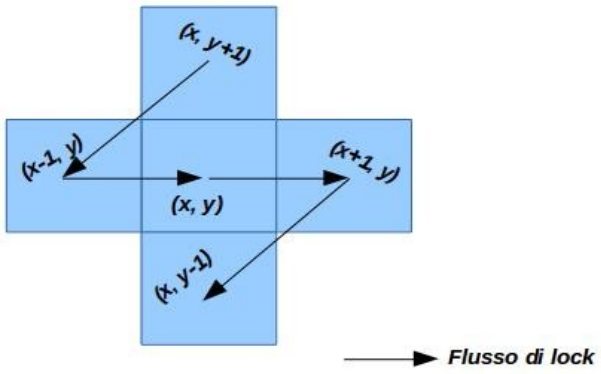

Illustrazione 3: Ordine con cui effettuare le lock

Ogni *worker* preleva le informazioni sulle dimensioni dell'area da elaborare dalla propria coda, elabora gli aggiornamenti e successivamente effettua l'inserimento di un argomento fittizio sulla coda del *collector*. Successivamente si mette in lettura sulla propria coda in

attesa che questa sia non vuota. Non ci sono informazioni che il *collector* ha necessità di ricevere dai singoli worker, ma ha bisogno di tenere traccia del numero di essi che terminano l'elaborazione, ecco perché si utilizza un argomento fittizio.

Altro problema affrontato è quello di dover assicurare che ogni essere vivente non venga mosso più di una volta per ogni *chronon.* Potrebbe, infatti, capitare che, a seguito di un movimento, un qualunque essere finisca nell'area di aggiornamento di un thread differente da quello precedente e dunque possa essere mosso una seconda volta. Per risolvere questo problema ho scelto di utilizzare una matrice di interi addizionale (chiamata *seen*) avente le stesse dimensioni del pianeta per poter impostare dei flag (rendendone positivi i valori corrispondenti) e tenere traccia delle celle che, essendo già state utilizzate, devono essere ignorate da ogni thread per tutto il *chronon.*

1.1.2| Il thread dispatcher

Il *dispatcher* analizza la matrice completa ed individua le informazioni sulle sottomatrici che i *worker* dovranno utilizzare. Per poterle incapsulare si è scelto di utilizzare una struttura dati apposita chiamata *square_t* che contiene le coordinate dei punti che individuano la diagonale delle aree da elaborare. E' infatti sufficiente avere queste due informazioni per poter portare effettuare l'aggiornamento.
Il dispatcher, una volta riempito un pool (che rimane invariato per l'intera esecuzione) effettua l'inserimento di *S* strutture, dove *S* è il numero di sottomatrici individuate. Successivamente anche il *dispatcher* si mette in attesa di poter prelevare un messaggio dalla propria coda.

1.1.3| Il thread collector

Il *collector* si assicura che tutti i *worker* terminino l'esecuzione, contando il numero di messaggi prelevabili dalla propria pila e aspettando che questo sia pari al numero dei lavoratori.
Fatto ciò, se il numero di chronon (che incrementa ad ogni aggiornamento) è pari al valore dopo il quale stampare la matrice, si occupa di aprire una connessione con il processo *visualizer* e di inviargli i dati. Il collector deve a questo punto anche azzerare tutti i valori della matrice seen. A questo punto effettua il controllo dei segnali (come discusso precedentemente).

1.2| Il processo visualizer

Il processo visualizer è, come detto in precedenza, un semplice *server* che aspetta connessioni da parte di *wator* e stampa il contenuto ricevuto. E' però anche il gestore della *socket* che viene impiegata per la comunicazione.
Sarà infatti delegata ad esso la necessità di creare il file fisico all'inizio della propria esecuzione e di assicurarsi che esso venga rimosso alla fine.
Se la *socket* fosse infatti presente al termine della terminazione del programma, causerebbe errori in fase di *binding* della connessione durante la successiva esecuzione.

1.3| Il protocollo di comunicazione

Il protocollo di comunicazione, come da richiesta, è stato sviluppato in modo da assicurare il minimo numero di byte scambiati. Esso deve assicurarsi che:

1. Il processo *wator* possa terminare *visualizer* con un apposito messaggio
2. Il processo *visualizer* riceva correttamente i dati della matrice.

I dati necessari per la stampa del pianeta sono:

1. I valori delle dimensioni
2. Tutti i valori contenuti nelle celle

I valori delle dimensioni sono rappresentabili attraverso un intero (*int*), avente una grandezza di *4 byte* per ogni dimensione. In totale dunque per inviarle entrambe, si impiegherebbero ben *8 byte*. Successivamente servirebbero *n byte*, dove *n* è il numero di celle della matrice.
Per ridurre lo spazio occupato dai valori delle due dimensioni, si costruisce un intero *raw* che può essere diviso in tre parti: un numero *j* di cifre per la rappresentazione di colonne, un numero *z* per quella delle righe e un valore (sempre in prima posizione) *k* (pari a *j*), utile per individuare la disgiunzione dei due precedenti.
In questo modo è possibile inviare due interi al costo di un singolo *integer*.

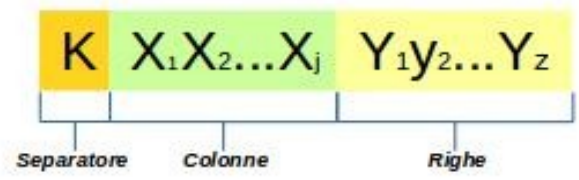

Illustrazione 4: Struttura del valore raw

Il valore *raw* è sempre un intero positivo, quindi è possibile renderlo negativo per segnalare la terminazione di *wator* e forzare la chiusura di *visualizer*.

Per ridurre invece il numero di *byte* trasferiti per la comunicazione della matrice, è stato adottato un semplicissimo algoritmo di compressione dei dati che si basa sulla sostituzione testuale di pattern statici. Il numero di valori assumibili da ogni cella della matrice è pari a tre (*'W', 'F', 'S'*), dunque ho costruito un vettore con tutte le combinazioni possibili dei valori individuati a gruppi di due elementi, che è pari a *3*3=9*!
In fase di compressione, *wator* sostituisce ogni pattern con l'indice del vettore ad esso corrispondente, mentre in fase di decompressione *visualizer* applica il procedimento inverso. In questo modo è possibile ridurre la stringa delle informazioni al più al 50%

rispetto all'originale!
Sarebbe possibile anche costruire il vettore dei pattern utilizzando gruppi di tre caratteri poiché il numero di combinazioni sarebbe comunque relativamente basso (*27*) e assicurerebbe una compressione dei dati al più al 20%.

Ho sviluppato anche un piccolo tool chiamato *compressiontest* scritto in bash che fornisce in output ogni informazione sulla compressione del pianeta che gli viene fornito in input.
Mi è stato utile in fase di sviluppo per assicurarmi della correttezza dell'algoritmo e per verificare i valori effettivi della compressione.

1.4| Le librerie

Per lo sviluppo del programma ho utilizzato due librerie: la libreria *libWator*, che forisce le funzioni utilizzate per l'interazione con il pianeta (dalle strutture di base, alle funzioni di movimento e di riproduzione) e la *libSynqueue* che fornisce l'implementazione delle code sincronizzate utilizzate per la comunicazione fra i thread.

Durante lo sviluppo del progetto ho dovuto effettuare delle modifiche alla libreria *libWator* per permettere il funzionamento dell'aggiornamento della matrice con la presenza dei thread. Inizialmente la funzione adibita a questo compito (*wator_update*) utilizzava un particolare escamotage per evitare di allocare la matrice *seen*: negativizzava il valore della matrice *dtime* (utilizzata per tenere traccia del numero di *chronon* restanti allo squalo prima morire di fame) per segnalare che quella posizione non dovesse essere più presa in considerazione durante la scansione. Ogni volta che, dunque, la funzione di aggiornamento individuava un valore negativo nella matrice *dtime* alla posizione *(i, j)*, ne ripristinava il valore e ignorava la casella corrispondente nella matrice del pianeta.
La correttezza dell'algoritmo era assicurata dal fatto che la scansione della matrice avvenisse in maniera sequenziale da sinistra verso destra unicamente da un thread, ma la sruttura a *farm* non mi ha successivamente permesso di mantenere questa configurazione e sono stato costretto ad adottare il metodo sopra discusso.

Le librerie vengono compilate in maniera corretta in seguito alla modifica del Makefile utilizzando il phony target *lib*.