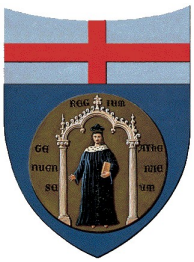
Università Degli Studi
Di Genova



Facoltà di Scienze Matematiche Fisiche Naturali
Corso di Laurea Magistrale in Informatica

Dessert

Una libreria per la simulazione a eventi discreti in .NET

Relatore:
Dott. Giovanni Lagorio
Correlatore:
Prof. Giovanni Chiola

Tesi magistrale di:
Alessio Parma
Matricola N° 3247806

Anno Accademico 2012/2013

Dedica

Questo "libercolo", che non raggiungerà nemmeno i venticinque lettori, è dedicato alla mia famiglia (mamma, papà e Matteo) e alla ragazza che amo, Marta. Se la scrittura delle restanti pagine è stata possibile, è gran parte merito loro.

Grazie alla mia famiglia ho potuto intraprendere questo percorso e, sempre grazie al loro sostegno, ho avuto i mezzi e il supporto per portarlo a termine. Inoltre, la loro compagnia e la loro presenza, nonché le buonissime torte della colazione di mia madre, le pizzette di mio padre e le strimpellate di mio fratello, hanno reso più lieve e felice il tempo passato a progettare e scrivere questo lavoro.

Anche Marta ha contribuito, con le sue dolci attenzioni, a rendere ancora più leggero il peso di questo compito: i suoi sorrisi sono stati di grande aiuto nei momenti più complicati, dove tra il caldo afoso e le varie difficoltà, la volontà tendeva a vacillare.

Non vi potrò mai ringraziare abbastanza per quanto avete fatto. Perciò, vi dedico questa relazione piena di strani concetti e nomi improbabili, e vi dico la cosa che ritengo più importante: vi voglio tanto bene...

"My mind," he said, "rebels at stagnation. Give me problems, give me work, give me the most abstruse cryptogram or the most intricate analysis, and I am in my own proper atmosphere. I can dispense then with artificial stimulants. But I abhor the dull routine of existence. I crave for mental exaltation. That is why I have chosen my own particular profession, or rather created it, for I am the only one in the world"

Sir Arthur Conan Doyle, The Sign of the Four

Indice

Capitolo 1

Introduzione

Nel corso degli anni sono state sviluppate molte librerie[1] a supporto della simulazione a eventi discreti, altrimenti nota come DES: Discrete Event Simulation. Al fine di essere meno prolissi, useremo frequentemente tale sigla nel corso della relazione. Tra le varie librerie dedicate alla DES, l'esempio più famoso è il linguaggio *Simula*[2], apparso negli anni '60 con lo scopo di offrire gli strumenti necessari per scrivere più facilmente il codice delle simulazioni. Dopodiché, molte altre librerie si sono susseguite [5]: tra le più famose e tuttora usate possiamo annoverare *ARENA* [6], la quale dispone di una notevole strumentazione e di un forte supporto commerciale, *NetSim* [7], dedicata alla simulazione di reti e di protocolli, e infine *SimPy* [8] che, servendosi del linguaggio Python, consente di scrivere simulazioni in maniera concisa ed elegante.

Il linguaggio SimPy ci è stato presentato nel corso di *Social and Peer to Peer Networks*, tenuto dai professori Marina Ribaudo [9] e Giovanni Chiola [10]. Tale linguaggio ci ha colpito per la sua semplicità d'uso, poiché in relativamente breve tempo, e senza particolari conoscenze pregresse sulla simulazione a eventi discreti, siamo riusciti a scrivere una simulazione mediamente complessa, nella quale abbiamo sperimentato gli effetti dell'applicazione dei campi di Galois [11] alla codifica dei pacchetti di rete (*linear network encoding* [12]).

Occorre porre enfasi sul fatto che, nonostante non avessimo mai scritto una simulazione, il linguaggio SimPy è stato facile e veloce da imparare, complice anche la chiarezza di Python. Tuttavia, ci siamo imbattuti in alcune proble-

[1] Talvolta, le librerie per simulazione a eventi discreti vengono definite *linguaggi*, in quanto le funzionalità da loro offerte impongono una sorta di *prassi* da seguire per ottenere ed eseguire una simulazione. Nel seguito della relazione faremo uso di entrambi i termini, anche se la maggior parte delle volte ci riferiremo a delle librerie e non a dei veri e propri linguaggi.

[2] Simula [1] è stato creato appositamente per la scrittura di simulazioni a eventi discreti; tuttavia, pur essendo nato solo per tale scopo, si è subito arricchito di funzionalità che per l'epoca erano innovative, tra le quali abbiamo le classi, i metodi virtuali, le *coroutine* [2] e un *garbage collector* [3]. Tali caratteristiche erano state aggiunte per facilitare e potenziare la scrittura delle simulazioni stesse, ma nel corso degli anni il loro principale beneficio è stato quello di *ispirare* altri progettisti di linguaggi di programmazione. Il caso più famoso è quello del creatore del C++, Bjarne Stroustrup, il quale ha apertamente dichiarato che il Simula è stato tra i maggiori ispiratori del C++ stesso [4].

matiche, che descriveremo in seguito, dall'osservazione delle quali è nata questa tesi.

In sezione 1.1, motiveremo quali ragioni ci abbiano spinti a realizzare Dessert, una libreria per la simulazione a eventi discreti per la piattaforma .NET [13]; inoltre, in sezione 1.2 descriveremo brevemente cosa sia la DES e a quali casi possa essere applicata con successo. Successivamente, in sezione 1.3, introdurremo la libreria SimPy, sul cui modello Dessert è basata. Infine, in sezione 1.4, faremo una panoramica dell'intera tesi.

1.1 Obiettivo della tesi

Come affermato nella premessa, la nostra esperienza con la libreria SimPy è stata positiva e siamo rimasti molto soddisfatti da come ci abbia rapidamente permesso di scrivere una simulazione mediamente complicata. Tuttavia, il fatto che essa sia basata su Python, oltre a portare un gran numero di vantaggi, primo fra tutti la notevole pulizia del codice, porta anche una serie di effetti collaterali che, sotto certi punti di vista, possono risultare grandi svantaggi.

In particolare, ci riferiamo al fatto che Python sia dinamicamente tipato e interpretato. Questi fatti non possono essere immediatamente etichettati come *difetti*, considerato che una parte importante del codice scritto quotidianamente a livello globale [14] ricade proprio in quella categoria (non solo in Python, ma anche in linguaggi più diffusi come JavaScript e PHP); tuttavia, è indubbio il fatto che un linguaggio con quelle caratteristiche avrà dei tempi di esecuzione non proprio ottimali e che la mancanza di ogni tipo di controllo statico rallenterà lo sviluppo del codice all'aumentare della dimensione del codice stesso.

Nel nostro caso specifico, ci siamo imbattuti in entrambi i difetti: abbiamo sia avuto difficoltà nella gestione del codice, vista la mancanza di controlli statici, sia avuto problemi a livello di prestazioni. In particolare, ciò che più ci ha colpito è stato il fatto che, nell'implementazione standard di Python (*CPython* [15]), esista un *lock* a livello globale [16] che impedisce al codice di sfruttare appieno le capacità offerte dai moderni sistemi con più unità di calcolo.

Nell'ambito della simulazione, come in un qualunque altro ambito di ricerca *empirica*, è necessario avere molti dati, raccolti in situazioni rilevanti, con i quali poter trarre decisioni significative. Pensando al semplice caso del lancio di una moneta, è ben noto il fatto che serve un cospicuo numero di lanci prima di avvicinarsi empiricamente alla nota probabilità di 0.5; analogamente, prima di poter trarre delle conclusioni da una simulazione è indubbiamente necessario eseguirla diverse volte, magari variando leggermente i parametri. Se una simulazione richiede molto tempo per essere eseguita, ottenere dati rilevanti sarà più complicato. Infatti, se il tempo richiesto fosse eccessivo, la simulazione sarebbe probabilmente eseguita un numero esiguo di volte e i risultati prodotti non sarebbero nemmeno affidabili. Inoltre, spesso occorre aumentare la *scala* della simulazione, con effetti potenzialmente disastrosi sui tempi di esecuzione.

Pertanto, il fattore *tempo* è di notevole importanza e, visto il comportamento non ottimale di SimPy, abbiamo deciso di indagare se fosse possibile creare

un'alternativa più efficiente, funzionante su un linguaggio fortemente tipato, compilato e con un supporto reale al *multithreading*. Tra i requisiti che ci siamo posti vi era quello di mantenere un'interfaccia il più simile possibile a quella di SimPy, poiché con essa siamo riusciti a lavorare in tutta comodità.

Da tali premesse nasce il progetto Dessert[3], con il quale abbiamo quasi totalmente raggiunto gli obiettivi preposti. Il nostro lavoro è stato composto dalle seguenti fasi:

1. Scelta di un linguaggio che rispettasse i nostri requisiti e che avesse un ottimo supporto per le coroutine; dopo diverse ricerche, esposte nella sezione 3.1, abbiamo scelto il linguaggio C# e la piattaforma .NET.

2. Scrittura di tutte le librerie necessarie per la creazione del simulatore, di cui parleremo nel capitolo 3.

3. Scrittura del simulatore stesso e preparazione della relativa documentazione.

4. Scrittura di esempi significativi con il quale poter dimostrare il funzionamento del nostro lavoro.

5. Preparazione di una serie di benchmark con i quali verificare il raggiungimento dei nostri obiettivi.

In corso d'opera ci è venuta un'ulteriore idea che non rientrava strettamente nei piani iniziali: la realizzazione di una sorta di *layer* con il quale poter prendere le simulazioni funzionanti su SimPy e farle eseguire sul nostro motore, unendo, in questo modo, l'eleganza delle simulazioni scritte in Python con la potenziale velocità del nostro motore, senza dover cambiare una singola riga di codice. I risultati ottenuti dalla nostra implementazione sperimentale, denominata *Armando*, saranno descritti nel capitolo 5.

Proseguiamo ora con alcuni cenni sulla simulazione a eventi discreti e con l'introduzione dei concetti principali sottostanti la libreria SimPy.

1.2 Simulazione a eventi discreti

In questa relazione sorvoleremo sugli aspetti prettamente teorici della DES, concentrandoci sugli aspetti tecnici legati all'implementazione di un motore per un simulatore. In ogni caso, è bene riportare alcuni di tali aspetti teorici, in modo che il lettore sappia quali tipi di situazioni possano essere oggetto di una simulazione; in particolare, useremo il documento in [17] come base per il nostro discorso.

Inoltre, traendo sempre spunto da tale documento, vedremo anche quali siano gli approcci *classici* al problema dell'implementazione di un simulatore e mostreremo quale sia il modello scelto da SimPy (e, di conseguenza, da Dessert).

[3]Si noti come il nome del nostro progetto, Dessert, tragga le sue origini proprio dall'acronimo di simulazione a eventi discreti, DES.

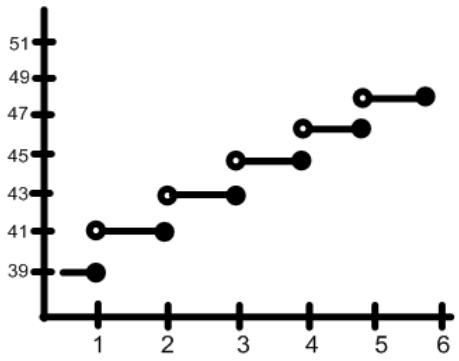


Figura 1.1: Grafico di una funzione a gradini.

1.2.1 Cosa può essere simulato

Come suggerisce il nome, è possibile simulare tutti quei sistemi in cui gli stati variano in modo *discreto* e non *continuo*. Tramite la DES è possibile modellare: la lunghezza di una coda, la quantità di merce conservata in un magazzino, il numero di operai al lavoro in un certo istante di tempo, etc. Per chiarire ulteriormente le idee, ricordiamo che il grafico di una variabile discreta al variare del tempo risulta essere riconducibile a quello di una funzione a gradini (figura 1.1).

Viceversa, una simulazione del livello dell'acqua in una rete fluviale ricadrebbe nella simulazione a eventi continui, poiché tutte le misure e le quantità coinvolte (livello dell'acqua, quantità di pioggia, velocità del flusso) sono continue; un altro classico esempio di sistema continuo è quello legato alla meteorologia: anche in quel caso, le misure rilevanti, come la temperatura, la pressione e l'umidità, sono continue.

Tra i sistemi classicamente modellati dalla DES possiamo annoverare le catene di immagazzinamento e trasporto dei materiali [18, 19, 20], la gestione delle code negli uffici pubblici [21, 22, 23] e i protocolli di rete [24, 25, 26]; inoltre, vi è tutta una serie di usi meno comuni, tra cui la simulazione del funzionamento dei circuiti elettronici [17].

1.2.2 Approcci alla simulazione

Una volta compresi quali siano i sistemi analizzabili con la DES, occorre chiedersi come sia possibile modellare un certo sistema e quali approcci possano risultare più o meno efficienti.

Per valutare i diversi paradigmi, prenderemo in considerazione un sistema semplice e consueto, quello di una banca. In questo caso gli eventi di rilievo sono:

- Ingresso di un cliente nella banca, circa ogni quarto d'ora.

- Aggiunta di un cliente a una delle code agli sportelli.
- Uscita di un cliente da una coda e inizio servizio; il servizio dura mediamente cinque minuti.
- Fine servizio allo sportello e uscita del cliente dalla banca.

Nelle sezioni seguenti vedremo come si possa realizzare un simulatore che sappia *muoversi* in mezzo a quel tipo di eventi e che sappia portare a termine la nostra simulazione.

1.2.2.1 Paradigma basato sulle attività

Il primo approccio al nostro problema potrebbe partire dal fatto che, secondo le nostre specifiche, abbiamo una granularità temporale degli eventi che si aggira attorno al minuto; infatti, gli ingressi avvengono in media ogni quindici minuti e i servizi agli sportelli durano, sempre in media, cinque minuti.

Pertanto, potremmo pensare di realizzare il nostro simulatore in modo tale che ogni X intervalli di tempo si fermi, osservi lo stato delle attività correnti e controlli il possibile avvenimento degli eventi a cui siamo interessati.

Nel nostro esempio la lunghezza dell'intervallo di tempo potrebbe essere nell'ordine dei dieci secondi, mentre le attività corrisponderebbero con il "generatore" di clienti in arrivo e con il servizio agli sportelli.

Un approccio simile, però, soffre di due difetti piuttosto evidenti. Il primo è dato dal fatto che il simulatore si ferma ogni X intervalli di tempo, indipendentemente dal fatto che vi siano o meno eventi da attivare. Il secondo dal fatto che non abbiamo usato da nessuna parte la nostra ipotesi iniziale, cioè, che gli eventi fossero collocati discretamente nel tempo.

Considerato che spesso le simulazioni sono piuttosto lunghe e popolate da un notevole numero di attività, si capisce subito che il paradigma basato sulle attività deve essere in qualche modo migliorato per poter essere applicato. Fortunatamente, è proprio il secondo difetto che ci suggerisce una possibile soluzione per il primo: sarebbe possibile trarre vantaggio dal fatto che gli eventi sono *discreti* per evitare un controllo periodico e *saltare* direttamente all'evento successivo? La risposta è si, come verrà spiegato nella sezione seguente. Occorre tuttavia notare che, nel caso della simulazione a eventi continui, non sia possibile servirsi di una scorciatoia simile; perciò, un simulatore di quel tipo dovrà seguire un approccio che non si discosta molto dal paradigma basato sulle attività [17].

1.2.2.2 Paradigma basato sugli eventi

Nel paradigma basato sugli eventi si riprende l'idea sottostante a quello basato sulle attività, applicando però la correzione secondo cui il tempo può essere direttamente fatto avanzare da un evento all'altro, senza fare controlli con una certa frequenza. Infatti, essendo gli eventi *discreti*, e supponendo di sapere

Listato 1.1 Sintesi del principio del paradigma basato su eventi.

```
// Rappresentazione sintetica di un evento.
// Il metodo Apply esegue l'attività
// dell'evento, come l'aggiunta in coda
// di un nuovo cliente della banca.
class Event {
    public double Time;
    public void Apply() { /* ... */ }
}

// Insieme ordinato dove vengono memorizzati gli eventi.
var eventSet = SortedSet<Event>();
AddSomeEvents(eventSet);
// Ciclo principale del simulatore.
while (eventSet.Count > 0)
    eventSet.RemoveMin().Apply();
```

quando avverrà l'evento successivo, sappiamo con certezza che nulla potrà accadere al nostro sistema nell'intervallo di tempo compreso tra il tempo attuale e il tempo del prossimo evento.

Il simulatore terrà traccia di tutti gli eventi e dei loro rispettivi tempi di esecuzione, memorizzandoli nel cosiddetto *event set* [17, 27]. A ogni iterazione, il simulatore prenderà semplicemente l'evento con il tempo minore, imposterà l'*orologio di sistema* in modo che sia uguale al tempo di quell'evento e lo attiverà. Nel listato 1.1 si delinea, tramite pseudo codice, quanto si è appena detto.

L'event set può essere implementato utilizzando diverse strutture dati; tra le più famose vi sono le code a priorità, le *skip list* [28] e gli alberi bilanciati. In ogni caso, molti studi sono stati condotti per analizzare quale fosse la struttura più adatta e molte altre strutture specializzate sono state proposte [29, 30].

L'approccio basato sugli eventi è visibilmente più efficiente, ma soprattutto più scalabile, di quello basato sulle attività. Infatti, la complessità algoritmica relativa al simulatore può essere abbassata notevolmente tramite la scelta di opportune strutture dati.

Supponendo che la simulazione abbia una durata D, che il numero medio di attività sia A e che l'intervallo di tempo sia X, la complessità indotta da un simulatore conforme al paradigma basato sulle attività è approssimativamente data da $O(A + \frac{D}{X}A)$, dove il primo addendo è dato dalla schedulazione delle attività e il secondo è dovuto al fatto che, in ogni istante di tempo, è necessario controllare che nessuna attività sia cambiata. Il fattore più pesante, però, è dato da D/X, poiché in genere D sarà molto grande e X piuttosto piccolo.

Nel caso della simulazione basata su eventi, invece, la scelta di una struttura dati ottimale può portare notevoli benefici. Supponendo di usare una coda a priorità basata su un albero binario bilanciato, la complessità per l'aggiunta

Listato 1.2 Implementazione del cliente secondo il paradigma basato su eventi.

```
1  class OnCustomerEntered : Event {
2      public void Apply() {
3          // La chiamata ad Enqueue, a tempo debito, aggiungerà
4          // un'istanza di OnCustomerOutOfQueue all'event set.
5          FindShortestQueue().Enqueue(TheCustomer);
6      }
7  }
8
9  class OnCustomerOutOfQueue : Event {
10     public void Apply() {
11         State = BeingServed;
12         EventSet.Add(new OnCustomerDone(), GetServiceTime());
13     }
14 }
15
16 class OnCustomerDone : Event {
17     public void Apply() {
18         State = Served;
19         RegisterStats();
20     }
21 }
```

di un elemento e la rimozione del minimo è in $\Theta(log_2(n))$, dove n è il numero degli elementi nella coda. Pertanto, riutilizzando i dati definiti in precedenza, la complessità diverrebbe $O(2A \cdot log_2(A))$ e avremmo dei miglioramenti solo se $2log_2(A) \ll 1 + {}^{D}/_{X}$, ma, come abbiamo detto in precedenza, ${}^{D}/_{X}$ è un numero elevato e quindi le nostre premesse sarebbero verificate.

1.2.2.3 Paradigma basato sui processi

Un ulteriore raffinamento al paradigma basato su eventi è stato quello di introdurre il concetto di processo, dove per processo si intende un flusso di esecuzione, cioè qualcosa che è più simile ai *thread* che non ai processi veri e propri del sistema operativo. Lo scopo era quello di rendere il codice della simulazione più modulare e leggibile, poiché il codice basato su eventi tendeva a non esserlo. A fine esemplificativo, e per far sì che il lettore abbia un'idea concreta di ciò di cui stiamo parlando, nel listato 1.2 è riportata una bozza dell'implementazione degli eventi relativi al cliente della banca, nell'ipotesi di stare scrivendo del codice basato su eventi.

Come visto nel listato, il paradigma basato su eventi porta alla scrittura di codice piuttosto tedioso e che tende a ricalcare male la realtà. Infatti, spesso la *vita* di ciascuna entità della simulazione può essere descritta da un insie-

Listato 1.3 Cliente della banca, secondo il paradigma basato su processi.

```
1 class Customer : Process {
2     public void Run() {
3         // Chiamata bloccante finché non è il suo turno.
4         FindShortestQueue().Enqueue(TheCustomer);
5         State = BeingServed;
6         Timeout(GetServiceTime());
7         State = Served;
8         RegisterStats();
9     }
10 }
```

me sequenziale di passi; nel caso specifico del cliente della banca, la sequenza potrebbe essere:

1. Entra in banca.
2. Inserisciti nella coda più breve.
3. Vieni servito allo sportello.
4. Esci dalla banca.

Già dal listato 1.2 si sarà notato che il paradigma a eventi porta alla creazione di un evento per ogni passo della vita dell'entità, eventi il cui collegamento viene perduto e che, come già detto, portano alla scrittura di molto codice.

Il paradigma basato sui processi, pur preservando l'idea chiave di quello basato su eventi, fa in modo che il codice scritto risulti pressoché identico a quello della sequenza numerata di passi, come mostriamo nel listato 1.3. Per ottenere tale obiettivo il simulatore fa uso di thread o di meccanismi simili, il che lo rende più lento rispetto a quelli basati sul paradigma a eventi. In ogni caso, i miglioramenti forniti alla leggibilità del codice e al suo design sorpassano di gran lunga i difetti prestazionali, ed è per questo motivo che ad oggi il paradigma basato sui processi è quello più frequentemente usato [17].

1.3 Libreria SimPy

Dopo aver introdotto alcuni concetti legati alla DES, possiamo passare a descrivere brevemente SimPy, alla quale sarà dedicato l'intero capitolo 2, e perché abbiamo ritenuto questa libreria così interessante.

Il linguaggio SimPy è basato sul paradigma a processi, che consente all'utente di scrivere codice leggibile e modulare, pur rimanendo discretamente efficiente. Tutto ciò è ancora migliorato dal fatto che il linguaggio con cui la libreria va usata è Python e che sono state fatte opportune scelte tecniche per *evitare* l'uso

dei thread, pur mantenendo una sorta di parallelismo virtuale. L'ultima caratteristica è quella che ci ha maggiormente colpito, perché le ragioni che hanno spinto gli sviluppatori di SimPy a portare avanti tale piano sono decisamente condivisibili.

Infatti, molte altre librerie per la DES sfruttano il meccanismo dei thread [31, 32], ma ciò implica svariati svantaggi tecnici e non; in particolare, abbiamo che:

- Il reale parallelismo richiede, sia da parte del simulatore, sia da parte dell'utente, l'uso di meccanismi di sincronizzazione come i lock, le barriere, etc etc. Ciò complica la scrittura del codice, dato che azioni anche molto semplici, come l'aggiunta di un elemento a una collezione, potrebbero portare a degli errori nel caso in cui la collezione sia condivisa tra più processi.

- Non tutte le piattaforme supportano i thread e, ove sono supportati, il loro comportamento è altamente variabile, soprattutto in termini di uso della memoria. Tale fatto mette a repentaglio la scalabilità della simulazione, che potrebbe dover coinvolgere quantità notevoli di processi.

- Una corretta implementazione basata su thread è difficile da raggiungere anche per i programmatori esperti. Viceversa, si vorrebbe far sì che anche utenti con poche conoscenze di programmazione possano avere la possibilità di utilizzare con successo il simulatore.

Gli sviluppatori di SimPy, consci di tali problematiche, hanno appunto deciso di rimuovere l'uso dei thread; il parallelismo viene ottenuto tramite un sapiente uso delle coroutine (presenti in Python sotto forma dei *generatori* [33]) e ciò permette di scrivere il codice con la garanzia che esso non sarà eseguito realmente in parallelo, ma in una maniera tale da emularlo. Un simile approccio porta ben due vantaggi immediati:

1. Il codice sarà più semplice e snello, poiché non sarà necessario l'uso delle strutture appositamente pensate per il reale parallelismo. Se due processi accedono alla stessa variabile, non si dovranno curare del fatto che l'altro processo stia accedendo o meno a tale variabile, perché SimPy garantisce che ciò non potrà avvenire.

2. Il design della simulazione potrà comunque *fingere* che il tutto avvenga in parallelo, come è giusto che sia. Pertanto, rimane la potenza espressiva della programmazione parallela, senza portarsi dietro alcuno svantaggio puramente tecnico.

Quindi, il nostro interesse per la libreria SimPy è scaturito sia dalla facilità con cui siamo riusciti a scrivere delle simulazioni, sia dalla sfida tecnica di poter realizzare una libreria equivalente che si servisse degli stessi *trucchi* per evitare l'uso dei thread. L'analisi di SimPy proseguirà nel capitolo 2, dove vedremo nel dettaglio ciò che la libreria offre e implementeremo realmente il semplice esempio della banca.

1.4 Panoramica dei prossimi capitoli

Nel capitolo 2 approfondiremo la conoscenza di SimPy e realizzeremo una breve simulazione al fine di prendere confidenza con tale libreria.

Successivamente, nel capitolo 3, vedremo in quali linguaggi fortemente tipati siano presenti le coroutine e motiveremo la nostra scelta di implementare il tutto sotto forma di libreria per la piattaforma .NET, dopo aver analizzato le offerte di C, C++, Java e .NET, alla luce anche del nostro obiettivo secondario di poter scrivere Armando, un layer per poter eseguire direttamente le simulazioni per SimPy. Sempre nello stesso capitolo, vedremo quale sia stato il restante lavoro preparatorio necessario alla scrittura di Dessert.

Dessert stessa sarà descritta nel capitolo 4, dove dettaglieremo le scelte implementative e le inevitabili differenze dal nostro modello, la libreria SimPy.

Nel capitolo 5 confronteremo le prestazioni fornite da Dessert, SimPy e Armando sottoponendole a simulazioni analoghe, con lo scopo di poter valutare se i nostri obiettivi iniziali siano stati raggiunti o meno.

Concluderemo, nel capitolo 6, esponendo anche alcune potenziali evoluzioni future.

Capitolo 2

Libreria SimPy

Consci del fatto che, nell'ambito della programmazione, una riga di codice valga più di tante parole, introdurremo la libreria SimPy proprio con l'esempio riportato nella pagina principale di tale progetto. Esso, mostrato nel listato 2.1, riassume in poche righe i concetti principali che abbiamo tanto apprezzato:

- Il codice è assolutamente pulito e chiaro. Anche chi non sa nulla di simulazioni e del paradigma a processi può intuire quale sarà il funzionamento dello script.
- Nessun uso dei costrutti legati alla multi programmazione, per rendere il codice snello e sicuro.

Dedicheremo questo capitolo alla descrizione della libreria, cosa che riteniamo necessaria per fornire al lettore le basi necessarie per comprendere le scelte fatte dalla nostra implementazione.

Inizieremo con l'illustrare, in sezione 2.1, le origini, gli autori e l'evoluzione del progetto SimPy, per poi passare, in sezione 2.2, a descrivere le idee chiave e i principi di design che stanno dietro tale libreria.

Continueremo con l'introduzione della classe `Environment`, in sezione 2.3, il punto di accesso alla libreria SimPy, proseguendo con gli strumenti messi a disposizione dello scrittore di simulazioni. Tratteremo degli eventi principali, in sezione 2.4, e delle *risorse*, in sezione 2.5, particolari strutture dati utili per emulare code, magazzini e, in generale, qualunque tipo di risorsa condivisa.

Concluderemo implementando, in sezione 2.6, l'esempio della banca introdotto nel capitolo 1.

2.1 Storia del progetto

Le prime versioni del progetto SimPy risalgono all'oramai lontano 2002 [34], quando fu pubblicato per la prima volta su *SourceForge*[1]. La libreria ha tratto

[1] SourceForge [35] è uno tra i più famosi siti di hosting per il codice e materiale relativo.

Listato 2.1 Semplice esempio d'uso di SimPy.

```
1  from simpy import *
2
3  def car(env):
4      while True:
5          print('Start parking at %d' % env.now)
6          parking_duration = 5
7          yield env.timeout(parking_duration)
8
9          print('Start driving at %d' % env.now)
10         trip_duration = 2
11         yield env.timeout(trip_duration)
12
13 env = simpy.Environment()
14 env.start(car(env))
15 env.run(until=15)
```

ispirazione dal già citato Simula e da un altro linguaggio dedicato alla simulazione, SIMSCRIPT[2]; dopodiché, nel corso degli anni, essa ha continuamente raffinato i concetti iniziali, sino ad arrivare alla versione 2.3, la quale è tuttora il rilascio *stabile* di tale progetto. A quel punto, la libreria non solo offriva le componenti per la simulazione, ma anche gli strumenti per ottenere grafici dai dati rilevati e per disegnare interfacce utente. Quella versione, rilasciata nel 2011, vanta utenti di eccellenza, tra cui la National Aeronautics and Space Administration (NASA) [37].

Tuttavia, di recente gli sviluppatori di SimPy hanno deciso di riscrivere completamente la loro libreria, al fine di potenziare l'espressività del loro linguaggio e di rendere il tutto eseguibile anche sulle versioni più recenti di Python; inoltre, è stato scelto di rimuovere tutte le componenti non strettamente legate alla simulazione, dato che, nel frattempo, altri strumenti decisamente più completi si sono imposti nel panorama dello sviluppo Python.

Il *nuovo* SimPy, marcato come versione 3, non è ancora stato rilasciato pubblicamente, ma noi abbiamo comunque deciso di basare Dessert proprio sul design di tale versione: se avessimo fatto il contrario, avremmo ottenuto un prodotto con un'interfaccia *datata* fin da subito. La nuova versione di SimPy ha posto maggiore enfasi sugli eventi, rendendo più intuitivo il loro uso e la loro combinazione in eventi più complessi tramite predicati logici, come la congiunzione e la disgiunzione, oppure tramite funzioni arbitrariamente defini-

[2] SIMSCRIPT [36], come Simula, è un linguaggio nato negli anni sessanta e che tuttora viene utilizzato per la DES. L'aspetto particolare di SIMSCRIPT è il fatto che il suo linguaggio è stato implementato in modo tale da renderlo il più simile possibile all'inglese scritto. Ad esempio, due variabili in virgola mobile possono essere definite con "`define a,b as double variables`", mentre si può assegnare un attributo a tutti gli oggetti di un certo tipo tramite codice simile a "`every customer has an cu.Arrival.time`".

Listato 2.2 Semplice esempio d'uso di SimPy 2.

```
1 from SimPy.Simulation import *
2
3 class Car(Process):
4     def run():
5         while True:
6             print('Start parking at %d' % now())
7             parking_duration = 5.0
8             yield hold, self, parking_duration
9
10            print('Start driving at %d' % now())
11            trip_duration = 2.0
12            yield hold, self, parking_duration
13
14 initialize()
15 c = Car()
16 activate(c, c.run(), at=0.0)
17 simulate(until=15.0)
```

te dall'utente. Soprattutto, la nuova interfaccia è solamente *somigliante* con la vecchia: per questa ragione, il nostro progetto non poteva non adattarsi, in previsione del futuro, al nuovo corso dello sviluppo.

Gli attuali sviluppatori del progetto risultano essere Stefan Scherfke [38] e Ontje Lünsdorf [39], mentre in passato hanno partecipato Tony Vignaux [40], professore emerito all'Università di Wellington, e Chang Chui, sul quale non abbiamo rinvenuto informazioni degne di nota.

Prima di concludere la sezione, vediamo, nel listato 2.2, come l'esempio in 2.1 risulti scritto per la versione 2 di SimPy. Il codice è leggermente più prolisso, poiché è necessario definire una classe contenente il processo in questione; inoltre, a ogni iterazione si restituisce una tupla composta dal comando e dai suoi relativi parametri, cosa che non rende esattamente intuitivo l'uso del simulatore. Infatti, in generale, non è immediatamente chiaro quali e quanti parametri siano necessari per ciascun comando e come sia possibile concatenarli (aspettare un evento e/o un altro).

2.2 Idee chiave

I concetti chiave relativi a SimPy sono già stati espressi in sezione 1.3, ma qui verranno ripresi e, ove necessario, ampliati. In particolare, vedremo meglio alcuni aspetti di base, come i processi e l'uso dei generatori, in modo da poterli dare come assodati nella parte restante del capitolo.

2.2.1 Uso del paradigma basato su processi

Un simulatore basato sul paradigma a processi risulta sia efficiente sia facile da usare per due ragioni fondamentali:

1. Si sfrutta il fatto che gli eventi siano *discreti*, per far sì che il motore del simulatore possa saltare direttamente da un evento all'altro, sapendo che tra i due eventi, per definizione, non può essere successo alcunché.

2. Si impone che l'utente organizzi il codice della simulazione in modo tale che ogni entità sia un processo a se stante; pertanto, non ci si focalizza sugli eventi, ma su come un processo interagisca con essi.

In SimPy qualunque *generatore* è candidato per essere un processo della simulazione. Come conseguenza, anche il semplice generatore riportato nel listato 2.1 è un processo valido di SimPy; sarà poi il simulatore ad aggiungere tutto il necessario per rendere la funzione un vero e proprio processo.

Le istanze dei processi sono restituite dal metodo `start` contenuto nella classe `Environment` e possono essere utilizzate sia per controllare lo stato del processo, sia per interagire con esso (usarlo come evento, lanciargli un segnale di interruzione, etc etc).

2.2.2 Uso dei generatori

I generatori sono il cuore di SimPy, ciò che lo rende unico nel suo settore. Tramite tale costrutto la libreria riesce a far sì che la simulazione possa essere progettata e pensata come se fosse eseguita in parallelo, ma con la sicurezza che in realtà sarà eseguita linearmente, alternando l'esecuzione dei vari processi.

L'espressione `yield` di Python viene usata nei processi per indicare i punti in cui il processo stesso è disposto a sospendere la propria esecuzione e lasciare il controllo al simulatore; il motore, in base alla condizione del sistema, potrà decidere se continuare con il medesimo processo, oppure se mandarne in esecuzione un altro, lasciando in pausa quello che ha lasciato il controllo.

Un generatore, dal punto di vista del linguaggio Python, è un oggetto che rappresenta, di fatto, una sequenza potenzialmente infinita di oggetti. Pertanto, i generatori possono tranquillamente essere usati dentro ai cicli, perché espongono i metodi necessari per poter essere manipolati in tale maniera.

Detto ciò, nel listato 2.3 proponiamo una versione estremamente succinta dell'implementazione del motore di SimPy: viene usato uno heap dove vengono inseriti i generatori e il rispettivo tempo in cui dovranno essere attivati. A ogni ciclo il motore prende il primo elemento dello heap e muove il generatore in avanti, facendo di fatto procedere la simulazione. Chiaramente, l'implementazione reale è decisamente più complicata di quanto mostrato, sebbene l'idea sottostante non si discosti molto.

Ritornando all'uso dei generatori da parte degli autori di simulazioni, si ha che, sempre tramite l'espressione `yield`, si indicano quei punti dove ci si aspetta un valore in entrata da parte del simulatore. Ad esempio, supponendo che il

Listato 2.3 Meccanismo centrale del motore di SimPy.

```
1  from heapq import *
2
3  class Process:
4      def __init__(self, gen, time):
5          self.gen = gen
6          self.time = time
7
8      def __cmp__(self, other):
9          return self.time.__cmp__(other.time)
10
11 class Environment:
12     def __init__(self):
13         self._agenda = []
14         self._now = 0.0
15
16     def start(self, gen):
17         p = Process(gen, self._now)
18         heappush(self._agenda, p)
19
20     def simulate(self):
21         while len(self._agenda) > 0:
22             p = self._agenda[0]
23             p.gen.next()
24             # Modifica dello heap a seconda
25             # dello stato di p dopo il next.
```

Listato 2.4 Uso dei generatori per ricevere valori.

```
1 def storeUser(env):
2     store = Store(env, capacity=1)
3     # Lo store è vuoto, per cui la chiamata
4     # sottostante sarà bloccante. Tuttavia,
5     # supponendo che qualcuno metta degli oggetti
6     # nello store, su item proprio ritroveremo uno
7     # di quegli oggetti.
8     item = yield store.get()
```

nostro processo si blocchi in attesa che sia il proprio turno per poter ricevere un oggetto da un magazzino, esso si aspetterà che, una volta rimesso in esecuzione, il simulatore gli abbia *passato* l'oggetto atteso. Vedremo meglio l'uso delle risorse in sezione 2.5, ma nel listato 2.4 è presente un semplice esempio con il quale si spera di chiarificare quanto appena detto.

Ciò che viene mostrato nell'esempio precedente è possibile soltanto perché i generatori di Python non espongono soltanto un metodo `next`, come tutti gli iteratori più comuni, ma anche i metodi `send` e `throw` [41]. In particolare, per Python `yield` è un'espressione, mentre per altri linguaggi è uno *statement*. Tali metodi, oltre a svolgere la stessa funzione di *avanzamento* compiuta da `next`, compiono un'ulteriore azione:

- `send` fa sì che il valore dell'espressione `yield` diventi quello dell'argomento passato alla funzione `send` stessa,
- `throw` fa sì che l'eccezione passata come argomento sia *iniettata* nel codice del generatore.

Come vedremo nel capitolo 4, la potenza dei generatori di Python non è comparabile all'offerta dei principali linguaggi per .NET e ciò ci ha portato a fare diverse scelte di design.

2.2.3 Uso degli "eventi" di SimPy

All'interno di SimPy troviamo un concetto semplice, ma al tempo stesso pratico e potente: un processo è letteralmente un *generatore di eventi*. Come si sarà notato nei brevi esempi finora introdotti, a ogni `yield` viene restituita un'istanza di un "evento": esso può essere un *timeout*, per indicare uno stop temporaneo, un processo, e molte altre cose.

Gli eventi, come è lecito aspettarsi, regolano il flusso della simulazione. Infatti, i processi possono gestire il proprio flusso di esecuzione solo tramite essi; ad esempio, un processo che si metta in coda per usufruire di una data risorsa dovrà necessariamente usare un evento apposito per arrestare la propria esecuzione finché non sarà il proprio turno.

Quindi, non è per nulla azzardato dire che, una volta compresa la gestione degli eventi, si sia in grado di scrivere simulazioni arbitrariamente complesse. Pertanto, essi saranno l'argomento principale del capitolo e, come si è già visto, compariranno in modo più o meno evidente in ciascuna sezione.

2.3 La classe `Environment`

Nella classe `Environment` di SimPy è situato, in un certo senso, il simulatore vero e proprio. Dalle istanze di tale classe è possibile avviare processi, avviare la simulazione stessa e controllare attributi come il tempo attuale; inoltre, un oggetto di tipo `Environment` è necessario per creare le risorse, che vedremo in sezione 2.5. Come suggerisce il nome, tale classe rappresenta l'ambiente di esecuzione delle simulazioni; pertanto, ora e nella parte restante del capitolo, ci riferiremo a un'istanza della classe `Environment` definendola come "ambiente", al fine di non essere eccessivamente pedanti e prolissi.

Si ritiene opportuno discutere dell'ambiente, prima degli altri elementi, poiché gli esempi successivi ne dovranno fare necessariamente uso. Infatti, anche la creazione dell'evento più semplice, il *timeout*, è effettuabile solo tramite esso, e, come abbiamo già detto, solo l'ambiente espone i metodi per avviare i processi.

2.3.1 Avvio di un processo

Un processo è semplicemente un generatore di eventi, che può essere avviato tramite il metodo `start` dell'ambiente, come mostrato nel listato 2.5. Tale metodo prende come input un generatore e lo restituisce opportunamente "decorato" in modo che esso risulti un vero e proprio processo della simulazione; le istanze ottenute in quella maniera potranno essere utilizzate come eventi, oppure potranno essere usate per chiedersi se un processo sia ancora attivo o meno.

Si vorrebbe porre enfasi sul fatto che l'ordine di avvio dei processi viene preservato dal simulatore. Quindi, sempre nel listato 2.5, avremo che i processi saranno eseguiti nello stesso ordine con cui sono stati avviati; in altre parole, la coda a priorità sottostante è stabile, ovvero, preserva l'ordine di inserimento degli elementi che hanno la stessa priorità.

Infine, va anche sottolineato il fatto che i processi possono essere avviati in qualunque momento, cioè, sia prima sia durante la simulazione. Il fatto che possano essere eseguiti nel mentre ci da la possibilità di implementare i cosiddetti processi "sorgente", il cui unico scopo è generare altri processi a dati intervalli di tempo; ricordando l'esempio della banca portato nell'introduzione, sarebbe necessario una sorgente per generare continuamente nuovi clienti.

2.3.2 Avvio di un processo "ritardatario"

Talvolta risulta utile avviare un processo con un certo ritardo; ad esempio, volendo modellare un guasto a una macchina, si potrebbero progettare due processi:

Listato 2.5 Uso del metodo `start` nell'ambiente di SimPy.

```
1 from simpy import *
2
3 # Risulta utile, se non necessario,
4 # passare un'istanza dell'ambiente ai processi.
5 def myProcess(env):
6     # Codice del processo, composto da una
7     # serie di yield alternate ad altro codice.
8
9 env = Environment()
10 p1 = env.start(myProcess(env))
11 p2 = env.start(myProcess(env))
12 p3 = env.start(myProcess(env))
13 # p1, p2 e p3 saranno eseguiti esattamente
14 # nell'ordine con cui sono stati avviati.
```

1. La macchina stessa, che esegue un certo insieme di operazioni che comportano dati intervalli di tempo.
2. Il guasto, il quale, periodicamente, potrebbe decidere se bloccare o meno una macchina.

Cercando di rimanere aderenti alla realtà, si ha che i guasti (per fortuna) solitamente non sovvengono nel primo periodo di funzionamento; quindi, sarebbe opportuno poter far partire il processo del guasto con un certo ritardo. Per quanto questo obiettivo si possa raggiungere in modo poco elegante con i timeout, che vedremo in seguito, esiste una prassi predefinita da SimPy ed è riportata nel listato 2.6. In pratica, invece di far partire direttamente il nostro processo, se ne fa partire un altro (`start_delayed`) che si occuperà si eseguirlo con il ritardo dato.

2.3.3 Avvio di una simulazione

Dopo aver aggiunto i processi che costituiranno lo stato iniziale della simulazione, essa può essere avviata tramite il metodo `run`. Esso può essere invocato in tre modi diversi:

1. Senza parametri, per indicare che la simulazione deve proseguire finché vi sono eventi. Se gli eventi sono infiniti, la simulazione non si fermerà e si potrà incorrere in problemi di *overflow* dovuto all'eccessivo avanzamento del tempo.
2. Con un parametro decimale, per impostare il tempo massimo della simulazione. Si ha la garanzia che la simulazione non andrà oltre il tempo indicato e che eventuali altri eventi schedulati esattamente per il tempo di fine, o successivamente, non saranno eseguiti.

Listato 2.6 Avvio di un processo con un ritardo dato.

```
1 from simpy import *
2 from simpy.util import *
3
4 def myProcess(env):
5     # Codice del processo...
6
7 env = Environment()
8 p1 = start_delayed(env, myProcess(env), 7)
9 p2 = start_delayed(env, myProcess(env), 3)
10 p3 = env.start(myProcess(env))
11 # Ordine di esecuzione: p3, p2, p1
```

3. Con un evento, per indicare che la simulazione dovrà proseguire finché l'evento dato non sarà accaduto. Anche in questo caso, valgono le considerazioni fatte al punto 1.

2.3.4 Attributi di `Environment`

Una volta che la simulazione è stata avviata, sarà possibile sapere il tempo attuale accedendo alla proprietà `now` e si potrà vedere il processo attivo tramite la proprietà `active_process`. A simulazione terminata, su `now` sarà possibile rinvenire l'ultimo istante in cui si è verificato un evento, utile per controllare che il tutto non si sia fermato in modo prematuro; ciò accade spesso ed è solamente dovuto a un'errata programmazione della simulazione stessa.

2.4 Panoramica degli eventi

Gli eventi, come già detto più volte, sono ciò che regola la vita dei processi; infatti, essi consistono nell'esecuzione di una serie di azioni alternate a delle attese di eventi. In particolare, nel caso di SimPy, le attese degli eventi coincidono le espressioni `yield`, così che sia facile individuare quali sono i punti in cui un processo arresta volontariamente la propria esecuzione.

Gli eventi messi a disposizione dalla libreria sono di varia natura, in quanto devono permettere di descrivere un gran numero di situazioni reali. Ad esempio, immaginando di dover scrivere una simulazione per un vecchio treno a carbone, troveremmo facilmente delle situazioni in cui il treno stesso si dovrebbe fermare in attesa di qualcosa o qualcuno:

- Finché gli operai non hanno caricato le scorte di carbone, il treno non può partire; gli operai possono essere, a loro volta, visti come un processo, la cui vita consiste nel caricare un certo treno. Pertanto, si è in un caso in cui un processo è anche un evento, il quale si *attiva* nel momento in cui il processo corrispondente ha finito il proprio compito.

- Un treno deve stare fermo in stazione per una certa quantità di tempo, fortemente dipendente dalla dimensione della stazione in cui esso si trova. In questo caso, stiamo trattando l'attesa subordinata a un evento che si attiverà dopo un certo intervallo di tempo.
- I treni sono dotati di un freno di emergenza, una volta tirato il quale il treno si deve fermare il prima possibile. Stiamo descrivendo un evento che può accadere in qualunque istante e che, nel momento in cui si attiva, ha priorità superiore agli altri eventi.
- Prima di partire da una stazione, il treno si deve assicurare che non vi siano passeggeri prossimi alla salita e che il capotreno sia di nuovo a bordo. Abbiamo appena introdotto il concetto di *combinazione logica* di eventi, la quale consiste nella creazione di eventi la cui attivazione è subordinata a quella degli eventi che si stanno combinando. Nel nostro esempio, abbiamo usato la congiunzione logica di due eventi, ma avremmo potuto usare anche predicati più complessi.

Tutte le situazioni sopra descritte sono rappresentabili all'interno di SimPy; compito di questa sezione sarà proprio quello di introdurre gli eventi di base, con i quali sarà già possibile descrivere un notevole numero di situazioni reali. Vi sono anche altri eventi di grande importanza, quelli legati all'uso delle risorse, che saranno trattati nella sezione successiva.

2.4.1 `Timeout`

Il passaggio del tempo è un elemento chiave della simulazione, in quanto viene spesso usato per rappresentare non soltanto delle pause, ma anche l'esecuzione di un certo compito. Supponendo di voler modellare un caso molto triviale come la cottura della pasta, si potrà ricorrere a un'attesa regolata dal tempo per simulare la cottura stessa; l'idea potrà essere complicata a piacere, tenendo conto del tipo di pasta e dei gusti del consumatore, ma il concetto di base sarà comunque lo stesso: simulare l'esecuzione di un dato compito aspettando il tempo necessario per il suo completamento.

All'interno di SimPy l'unico strumento a disposizione dell'utente per far scorrere il tempo è l'evento `Timeout`, il quale può essere configurato in modo che si attivi con un ritardo dato e restituisca un certo valore all'attivazione. Le istanze di quell'evento possono essere generate da un oggetto di tipo `Environment`, come molti altri eventi che vedremo in seguito.

All'interno del simulatore il tempo è diviso in unità, che non sono strettamente secondi, minuti o altre usuali misure del tempo. Il significato dell'unità di tempo deve essere stabilito da chi scrive la simulazione, in base a ciò che si sta modellando. Per esempio, nel caso di un protocollo di rete, avrà molto senso stabilire un'unità di tempo che valga un millisecondo, mentre nel caso della gestione di un magazzino si potrà salire al minuto.

I listati 2.7 e 2.9 mostrano due esempi d'uso dei *timeout*; nel primo caso, un processo "cuoco" deve preparare della pasta avendo un solo fornello, quindi

Listato 2.7 Simulazione della cottura della pasta.

```
1  from random import *
2  from simpy import *
3
4  rand = Random(21)
5  avgCookTime = 10 # Minuti
6  stdCookTime = 1 # Minuti
7  simTime = 50 # Minuti
8
9  def pastaCook(env):
10     normal = rand.normalvariate
11     while True:
12         cookTime = normal(avgCookTime, stdCookTime)
13         print("Pasta in cottura per %g minuti" % cookTime)
14         yield env.timeout(cookTime)
15         if cookTime < avgCookTime - stdCookTime:
16             print("Pasta poco cotta!")
17         elif cookTime > avgCookTime + stdCookTime:
18             print("Pasta troppo cotta...")
19         else:
20             print("Pasta ben cotta!!!")
21
22 env = Environment()
23 env.start(pastaCook(env))
24 env.run(simTime)
```

dovrà far cuocere sequenzialmente le proprie pentole. Notiamo come l'atto di cuocere venga rappresentato con un timeout variabile e come si sia scelto il minuto come unità di tempo. Un possibile output di tale script è riportato nel listato 2.8.

Il secondo esempio, un fantasioso tiro al bersaglio, evidenzia come gli eventi possano essere passati tra i metodi e come si possa associare un valore a un timeout. Nello script viene definito un generatore di bersagli, creati con il ritardo a cui appariranno e con ciò che essi rappresentano (un alieno, ad esempio); tali eventi verranno usati da un processo tiratore, con i quali saprà il momento in cui i bersagli saranno visibili. Nel listato 2.10 riportiamo un output generabile con quello script.

2.4.2 Process

Poter utilizzare i processi come eventi da la possibilità di descrivere immediatamente tutta una serie di situazioni molto comuni, come il caricamento di una macchina da parte di un operaio, l'attesa di un panettiere che richiede il pro-

Listato 2.8 Possibile output dell'esempio in 2.7.

```
Pasta in cottura per 8.14724 minuti
Pasta poco cotta!
Pasta in cottura per 10.4446 minuti
Pasta ben cotta!!!
Pasta in cottura per 7.65061 minuti
Pasta poco cotta!
Pasta in cottura per 11.0833 minuti
Pasta troppo cotta...
Pasta in cottura per 10.0114 minuti
Pasta ben cotta!!!
Pasta in cottura per 8.64488 minuti
```

Listato 2.9 Simulazione di un fantasioso tiro al bersaglio.

```
from random import *
from simpy import *

rand = Random(21)
hitProb = 0.7
simTime = 100 # Secondi

def newTarget(env):
    delay = rand.uniform(1, 20)
    target = rand.choice(["Alieno", "Pollo", "Unicorno"])
    return env.timeout(delay, target)

def shooter(env):
    while True:
        tgt = yield newTarget(env)
        hit = rand.random();
        if hit < hitProb:
            print("%g: %s colpito, si!" % (env.now, tgt))
        else:
            print("%g: %s mancato, no..." % (env.now, tgt))

env = Environment()
env.start(shooter(env))
env.run(simTime)
```

Listato 2.10 Possibile output dell'esempio in 2.9.

```
4.13404: Unicorno colpito, si!
14.2369: Alieno mancato, no...
30.5862: Pollo colpito, si!
36.0713: Alieno colpito, si!
48.1931: Alieno mancato, no...
53.6059: Alieno colpito, si!
73.5596: Unicorno mancato, no...
86.2615: Alieno colpito, si!
96.715: Alieno mancato, no...
```

dotto dell'impastatrice o di una massaia che attende il lavaggio dei panni per poterli stendere ad asciugare.

L'evento corrispondente a un dato processo viene restituito dal metodo start di `Environment`; tale oggetto, oltre a poter essere usato come evento, può essere utilizzato per recuperare informazioni aggiuntive sul processo tramite, ad esempio, le proprietà `is_alive`, che indica se esso sia ancora in esecuzione, e `target`, sul quale può essere ritrovato l'evento che il processo sta aspettando.

Nel listato 2.11 vediamo come un processo, una macchina, attenda che un secondo processo, un operaio, la carichi e le indichi l'operazione da eseguire; l'output di tale script può essere visto nel listato 2.12. Il codice riportato dovrebbe risultare intuitivo, all'infuori del comando `exit`, di cui non abbiamo ancora parlato: esso indica solamente al simulatore che il processo ha terminato la propria esecuzione e che si deve inoltrare un valore dato a tutti quelli che erano in sua attesa. Nel caso dell'esempio, il valore da inoltrare è il tipo del comando, il quale viene ricevuto dalla macchina in attesa del completamento del lavoro dell'operaio.

2.4.3 `Event`

Mentre per gli eventi descritti finora, `Timeout` e `Process`, l'attivazione era implicita, cioè, non era compito dell'utente scatenarli, in quanto essi si sarebbero attivati da soli[3], ora vediamo un nuovo tipo, `Event`, che si comporta in modo differente; infatti, esso espone due metodi, `succeed` e `fail`, con i quali è possibile controllare l'attivazione dell'evento.

In particolare, entrambi possono ricevere un valore in input, che verrà gestito in maniera diversa da ciascuno di loro: `succeed` farà si che l'evento sia attivato con successo e il valore specificato sarà inoltrato a tutti i processi che erano in attesa di quell'evento; al contrario, `fail`, oltre a marcare l'evento come fallito, invierà un'eccezione a tutti coloro che attendevano tale evento. L'eccezione

[3]Timeout viene attivato allo scadere dell'intervallo specificato, mentre Process è attivato al termine dell'esecuzione del processo stesso.

Listato 2.11 Simulazione del caricamento di una macchina.

```
1  from random import *
2  from simpy import *
3
4  rand = Random(21)
5  loadTime = 5 # Minuti
6  workTime = 25 # Minuti
7  simTime = 100 # Minuti
8
9  def worker(env):
10     print("%g: Carico la macchina..." % env.now)
11     yield env.timeout(loadTime)
12     env.exit(rand.choice(["A", "B", "C"]))
13
14 def machine(env):
15     while True:
16         cmd = yield env.start(worker(env))
17         print("%g: Eseguo il compito %s" % (env.now, cmd))
18         yield env.timeout(workTime)
19
20 env = Environment()
21 env.start(machine(env))
22 env.run(simTime)
```

Listato 2.12 Possibile output dell'esempio in 2.11.

```
1 0: Carico la macchina...
2 5: Eseguo il compito A
3 30: Carico la macchina...
4 35: Eseguo il compito C
5 60: Carico la macchina...
6 65: Eseguo il compito B
7 90: Carico la macchina...
8 95: Eseguo il compito B
```

Listato 2.13 Esempio d'uso delle istanze di Event.

```
1  from simpy import *
2
3  def doSucceed(env, ev):
4      yield env.timeout(5)
5      ev.succeed("SI :)")
6
7  def doFail(env, ev):
8      yield env.timeout(5)
9      ev.fail(Exception())
10
11 def proc(env):
12     ev1 = env.event()
13     env.start(doSucceed(env, ev1))
14     print(yield ev1)
15     ev2 = env.event()
16     env.start(doFail(env, ev2))
17     try: yield ev2
18     except Exception: print("NO :(")
19
20 env = Environment()
21 env.start(proc(env))
22 env.simulate()
```

inviata da `fail` corrisponde con il valore specificato; pertanto, esso dovrà essere un'istanza di un'eccezione Python.

Nel listato 2.13 osserviamo come gli eventi possano essere creati dal metodo `event` di `Environment` e come il loro uso rispecchi quanto abbiamo appena descritto. La nostra piccola simulazione esemplificativa consiste di tre processi, due dedicati all'attivazione degli eventi e uno dedicato alla creazione e all'attesa degli eventi stessi. La parte interessante è l'uso del costrutto `try-except`, con il quale si gestisce il fallimento del secondo evento. L'eccezione inviata è effettivamente un'eccezione come tutte le altre; pertanto, se non la avessimo catturata, si sarebbe propagata al resto del programma, aiutando l'utente a non dimenticare di gestire il fallimento di un evento. Come il lettore avrà intuito, l'output del listato in questione è semplicemente "`SI :)`", seguito da "`NO :(`".

La classe `Event`, così come `Timeout` e `Process`, espone la possibilità di aggiungere delle funzioni, dette *callback*, che verranno invocate nel momento in cui l'evento sarà attivato. Esse possono essere aggiunte direttamente all'attributo `callbacks` e devono accettare un solo parametro, l'evento sul quale esse sono state registrate.

Un esempio d'uso delle *callback* è mostrato nel listato 2.14, dove modifichiamo leggermente l'esempio in 2.13 e aggiungiamo la definizione di una callback,

Listato 2.14 Esempio d'uso delle *callback*.

```
1  from simpy import *
2
3  def doFail(env, ev):
4      yield env.timeout(5)
5      ev.fail(Exception("NO"))
6
7  def myCallback(ev):
8      print("Successo: '%s'; Valore: '%s'" % (ev.ok, ev.value))
9
10 def proc(env):
11     ev1 = env.timeout(7, "SI")
12     ev1.callbacks = [myCallback]
13     yield ev1
14     ev2 = env.event()
15     ev2.callbacks = [myCallback]
16     env.start(doFail(env, ev2))
17     try: yield ev2
18     except Exception: pass
19
20 env = Environment()
21 env.start(proc(env))
22 env.run()
```

`myCallback`, con la necessaria registrazione alle righe 12 e 15. Nell'esempio, abbiamo sostituito il primo evento con un timeout, in modo da chiarificare il fatto che anche su tali eventi sono registrabili delle funzioni. Considerato che i timeout hanno sempre successo, e che il secondo evento viene fatto fallire, abbiamo che l'output dell'esempio sarà "`Successo: 'True'; Valore: 'SI'`", seguito da "`Successo: 'False'; Valore: 'NO'`".

2.4.4 `Condition`

Talvolta è necessario combinare gli eventi di base in modo tale da ottenere funzionalità più complesse. Un caso molto frequente sono le politiche di timeout, dove si attende il verificarsi di un certo evento, ma non oltre un dato intervallo di tempo; altri casi comuni prevedono l'attesa di tutti gli eventi di un dato insieme, oppure l'attesa di almeno un evento. Tuttavia, vi sono anche altri casi, magari meno frequenti, per i quali è necessario vincolare l'attesa a condizioni più complesse e imprevedibili, come predicati del tipo "due eventi si devono essere verificati con valore sette, oppure un dato intervallo di tempo deve essere passato e un processo deve aver concluso il proprio lavoro".

La libreria SimPy offre il supporto a entrambe le casistiche, cioè, quelle

frequenti e quelle più rare; chiaramente, il tutto è stato progettato in modo tale che le combinazioni più frequenti siano più facili da realizzare, mentre quelle più complicate e non comuni richiedano più lavoro da parte dell'utente.

Le casistiche comuni, per le quali vi è maggiore supporto, sono la disgiunzione e la congiunzione di eventi. Con esse è possibile coprire molti casi d'uso, come le politiche di timeout, l'attesa di uno o tutti gli elementi di un dato insieme e qualunque tipo di espressione logica ottenibile con tali operatori. La classe `Event` implementa opportunamente gli operatori di congiunzione (`&`) e disgiunzione (`|`), in modo tale che sia intuitivo combinare due eventi tra loro; inoltre, sono messi a disposizione due metodi di `Environment`, `any_of` e `all_of`, per gestire, rispettivamente, l'attesa di uno o tutti gli eventi di un insieme.

Condizioni più complicate possono essere realizzate tramite l'uso della classe `Condition` e la definizione di opportune funzioni per la loro valutazione. La costruzione di un'istanza di tale classe richiede, oltre a una funzione, anche la lista con tutti gli eventi coinvolti. In particolare, la segnatura di una funzione per la valutazione deve prevedere:

- Un valore di ritorno booleano, il quale indichi se la condizione sia stata verificata o meno.
- Una lista con tutti gli eventi coinvolti nella condizione.
- Un dizionario, le cui chiavi sono gli eventi verificati e i cui elementi sono i valori associati a tali eventi.

La funzione di valutazione verrà invocata ogniqualvolta un evento, fra quelli specificati, si verifichi e la condizione sarà verificata, a sua volta, nel momento in cui la funzione restituirà il valore vero.

I listati 2.15 e 2.16 mostrano dei semplici esempi d'uso delle condizioni, con il relativo output. Tra gli elementi degni di nota, possiamo annoverare l'uso dei metodi `all_of` e `any_of`, per gestire insiemi di eventi, oltre alla definizione della funzione di valutazione `customEval` all'interno del processo `conditionTester`.

2.4.5 `Interrupt`

In base a quello che abbiamo visto finora, un processo può arrestare la propria esecuzione e attendere diversi tipi di eventi, come i timeout, altri processi, condizioni, etc etc; tuttavia, una volta che il processo si è volontariamente arrestato, solo l'attivazione dell'evento atteso può riattivarlo. Il vincolo appena descritto è un po' troppo restrittivo, in quanto vi sono casi, facilmente immaginabili, che non possono essere descritti al meglio.

Alcuni esempi molto comuni sono rappresentati dai vigili del fuoco e dalle unità di protezione civile, che devono interrompere qualunque attività stiano eseguendo in caso di chiamate di emergenza; inoltre, tutti quei processi che potrebbero richiedere la gestione di eventi eccezionali e rari, come la rottura di uno strumento di un operaio, sarebbero tagliati fuori dal vincolo sopra esposto.

Come è facile aspettarsi, SimPy offre quanto necessario per abbattere il vincolo e rappresentare ulteriori casistiche; il concetto introdotto è quello di

Listato 2.15 Esempio d'uso della combinazione di eventi.

```
1  from simpy import *
2  from simpy.util import *
3
4  def aProcess(env):
5      yield env.timeout(7)
6      env.exit("VAL_P")
7
8  def conditionTester(env):
9      aProc = env.start(aProcess(env), "P")
10     res = yield all_of([env.timeout(5, "VAL_T", "T"), aProc])
11     print("ALL: %s" % res)
12
13     aProc = env.start(aProcess(env), "P")
14     res = yield any_of([env.timeout(5, "VAL_T", "T"), aProc])
15     print("ANY: %s" % res)
16
17     def customEval(events, values):
18         return len(events) == len(values) \
19            and values[events[0]] == "VAL_T" \
20            and values[events[1]] == "VAL_P"
21
22     aProc = env.start(aProcess(env), "P")
23     aTime = env.timeout(5, "VAL_T", "T")
24     res = yield Condition(env, customEval, [aTime, aProc])
25     print("CUSTOM: %s" % res)
26
27 env = Environment()
28 env.start(conditionTester(env))
29 env.run()
```

Listato 2.16 Output dello script in 2.15.

```
1 ALL: {P: 'VAL_P', T: 'VAL_T'}
2 ANY: {T: 'VAL_T'}
3 CUSTOM: {T: 'VAL_T', P: 'VAL_P'}
```

interrupt, con il quale possono essere riattivati i processi in attesa di altri eventi. Un processo, inteso come istanza di `Process`, può essere interrotto tramite il metodo `interrupt`, al quale può essere opzionalmente passato un valore che, a sua volta, sarà inoltrato al processo stesso.

Una volta interrotto, un processo sarà immediatamente risvegliato e riceverà un'eccezione di tipo `Interrupt`. Come nel caso del fallimento di `Event`, l'eccezione in questione è una reale eccezione Python, che, se non opportunamente catturata, farà interrompere l'esecuzione di tutta la simulazione. Nel momento in cui viene risvegliato, un processo può scegliere se ignorare il segnale, riprendendo l'attesa del precedente evento, o se cambiare le operazioni future.

Il listato 2.17 presenta un'applicazione del concetto di interruzione direttamente su un caso concreto, l'andamento di un treno. Esso procede tra una stazione e l'altra, trascorrendo del tempo in viaggio e del tempo in stazione, aspettando che i passeggeri salgano a bordo. Tuttavia, in qualunque momento un passeggero potrebbe tirare il freno di emergenza: in quel caso, il treno deve arrestare *immediatamente* il proprio moto. Il caso appena descritto calza a pennello con l'uso più comune degli interrupt, nel quale si descrive la reazione di un processo alle situazioni di emergenza.

Relativamente all'esempio proposto, il processo del treno monitora, tramite il costrutto `try-except`, il possibile arrivo di nuove interruzioni; esse sono causate dal processo `emergencyBrake`, il quale, tramite un riferimento al processo del treno, lo interrompe dopo che un certo intervallo di tempo è trascorso. Un possibile output dell'esempio è riportato nel listato 2.18.

2.5 Panoramica delle risorse

Finora, abbiamo introdotto il concetto di processo e di evento all'interno di SimPy, descrivendo come un processo possa attendere un evento e come più processi possano interagire tra di loro, anche tramite l'uso delle interruzioni. Tuttavia, quegli strumenti non sono ancora sufficienti per rappresentare situazioni reali molto comuni, per le quali un simulatore deve necessariamente offrire supporto.

Infatti, servendoci solo dei concetti introdotti, non potremmo assolutamente rappresentare risorse con una capacità prefissata e le rispettive code d'attesa. Tra le situazioni che possono essere descritte con quegli strumenti abbiamo le code negli uffici pubblici, i magazzini, le pompe di benzina e molto altro.

Quindi, servirebbero delle risorse che diventino *bloccanti* nel caso in cui l'operazione richiesta vada contro il vincolo di capacità. Per esempio, supponendo di aggiungere un elemento a un magazzino pieno, vorremmo che il nostro processo fosse bloccato in attesa del suo svuotamento.

Pertanto, dedicheremo questa sezione alla descrizione delle risorse messe a disposizione da SimPy; vedremo le risorse semplici, le quali hanno una capacità limitata e intera, per poi passare agli *store* e ai *container*, con i quali poter simulare vere e proprie unità di immagazzinamento.

Listato 2.17 Simulazione del freno di emergenza di un treno.

```
1  from random import *
2  from simpy import *
3
4  rand = Random(21)
5  avgTravelTime = 20.0 # Minuti
6  breakTime = 50 # Minuti
7
8  def train(env):
9      while True:
10         try:
11             time = rand.expovariate(1.0/avgTravelTime)
12             print("Treno in viaggio per %g minuti" % time)
13             yield env.timeout(time)
14             print("Arrivo in stazione, attesa passeggeri")
15             yield env.timeout(2)
16         except Interrupt, i:
17             print("Al minuto %g: %s" % (env.now, i.cause))
18             break
19
20 def emergencyBrake(env, tr):
21     yield env.timeout(breakTime)
22     tr.interrupt("FRENO EMERGENZA")
23
24 env = Environment()
25 tr = env.start(train(env))
26 env.start(emergencyBrake(env, tr))
27 env.run()
```

Listato 2.18 Possibile output dello script in 2.17.

```
1 Treno in viaggio per 3.60526 minuti
2 Arrivo in stazione, attesa passeggeri
3 Treno in viaggio per 23.4086 minuti
4 Arrivo in stazione, attesa passeggeri
5 Treno in viaggio per 20.1572 minuti
6 Al minuto 50: FRENO EMERGENZA
```

2.5.1 `Resource`, `PriorityResource` e `PreemptiveResource`

Le risorse, modellate dalla classe `Resource`, hanno una capacità di tipo intero, la quale deve essere specificata dall'utente; inoltre, esse espongono un metodo, `request`, con il quale un processo può richiedere un'unità della risorsa stessa. Nel caso non vi siano unità libere, il processo sarà messo in coda e svegliato appena possibile.

Quindi, con le risorse possiamo *controllare* e simulare gli accessi a risorse condivisibili da un numero massimo di entità. Cercando sempre di rifarci a casi concreti e di tutti i giorni, le risorse ci aiutano a modellare ambienti come i benzinai, che hanno un numero prefissato di pompe per la benzina, e gli uffici pubblici, che offrono ai clienti solo un certo numero di sportelli.

Un altro esempio modellabile tramite le risorse è rappresentato dai bagni pubblici, dove spesso troviamo un determinato numero di bagni per le donne e per gli uomini; tale "sistema" è sinteticamente rappresentato nel listato 2.19, dove un "generatore di persone" crea, a intervalli di tempo casuali, delle persone e le avvia verso i bagni. Una persona può essere donna o uomo e, considerato che vi è un solo bagno per tipo, molto probabilmente dovrà passare del tempo in coda. I bagni sono modellati con delle risorse a capacità uno, al fine di aumentare la possibilità che una persona debba aspettare; se si dovessero simulare dei bagni "reali", cioè, con più posti a disposizione, sarebbe sufficiente incrementare opportunamente la capacità delle risorse, in modo tale che sia uguale al numero dei posti. Un possibile output generato dall'esecuzione dello script in 2.19 è riportato nel listato 2.20.

`PriorityResource` Ritornando alla descrizione degli altri tipi di risorse messe a disposizione da SimPy, occorre fare alcune considerazioni pratiche: negli esempi fatti finora, le code erano gestite in modo tale che tutti i processi avessero la stessa priorità; pertanto, l'ordine di elaborazione di una coda dipendeva unicamente dall'ordine di arrivo. Però, vi sono casi concreti in cui la posizione in coda non è determinata solo dall'ordine di arrivo, ma anche da una *priorità* che può essere arbitrariamente assegnata; i pronto soccorso all'interno degli ospedali italiani, ad esempio, seguono una politica simile, assegnando un *colore* alla gravità del paziente: un codice rosso avrà priorità sui verdi, anche se questi erano arrivati molto prima.

Quindi, al fine di poter modellare correttamente situazioni simili, è opportuno introdurre una nuova classe, `PriorityResource`. L'uso di tale classe è praticamente identico a quello di `Resource`, escluso il fatto che il metodo `request` accetta un parametro aggiuntivo, la priorità; essa viene gestita in modo tale che le priorità numericamente minori abbiano precedenza sulle altre, come viene fatto nei processi UNIX [42].

Al fine di chiarificare quanto appena descritto, abbiamo creato una semplice simulazione del funzionamento del pronto soccorso italiano, mostrata nel listato 2.21. I processi, o meglio, i pazienti, accedono alla risorsa ospedale appena sono stati avviati; nel caso la risorsa sia libera, essi vengono curati, altrimenti rimangono in attesa. La priorità viene specificata direttamente nel metodo

Listato 2.19 Simulazione del funzionamento dei bagni pubblici.

```
from random import *
from simpy import *

rand = Random(21)
avgPersonArrival, avgTimeInToilet = 1.0, 5.0 # Minuti
simTime = 10 # Minuti
exponential = rand.expovariate

def person(env, gender, toilet):
    with toilet.request() as req:
        yield req
        print("%.2f: %s --> Bagno" % (env.now, gender))
        yield env.timeout(exponential(1.0/avgTimeInToilet))
        print("%.2f: %s <-- Bagno" % (env.now, gender))

def personGenerator(env):
    womenTt, menTt = Resource(env, 1), Resource(env, 1)
    id = 0
    while True:
        isWoman = rand.random() < 0.5
        gender = ("Donna" if isWoman else "Uomo") + str(id)
        toilet = womenTt if isWoman else menTt
        id += 1
        print("%.2f: %s in coda" % (env.now, gender))
        env.start(person(env, gender, toilet))
        yield env.timeout(exponential(1.0/avgPersonArrival))

env = Environment()
env.start(personGenerator(env))
env.run(simTime)
```

Listato 2.20 Possibile output dello script in 2.19.

```
0.00: Uomo0 in coda
0.00: Uomo0 --> Bagno
0.02: Donna1 in coda
0.02: Donna1 --> Bagno
0.40: Donna2 in coda
0.42: Uomo3 in coda
0.49: Donna1 <-- Bagno
0.49: Donna2 --> Bagno
1.84: Donna2 <-- Bagno
4.03: Uomo0 <-- Bagno
```

Listato 2.21 Simulazione della gestione dei pazienti in un pronto soccorso.

```
1  from simpy import *
2
3  # Priorita' decrescenti, piu' una priorita' e' bassa
4  # prima il processo sara' posizionato nella coda.
5  RED, YELLOW, GREEN = 0, 1, 2
6
7  def person(env, name, code, hospital):
8      with hospital.request(code) as req:
9          yield req
10         print("%s viene curato..." % name)
11
12 env = Environment()
13 hospital = PriorityResource(env, capacity=2)
14 env.start(person(env, "Pino", YELLOW, hospital))
15 env.start(person(env, "Gino", GREEN, hospital))
16 env.start(person(env, "Nino", GREEN, hospital))
17 env.start(person(env, "Dino", YELLOW, hospital))
18 env.start(person(env, "Tino", RED, hospital))
19 env.run()
```

`request`, dove si fa uso del codice assegnato. L'ospedale è piuttosto piccolo, poiché ha solo due posti, e i pazienti arrivano nel seguente ordine:

1. Pino, codice giallo;
2. Gino, codice verde;
3. Nino, codice verde;
4. Dino, codice giallo;
5. Tino, codice rosso.

I primi due pazienti, Pino e Gino, vengono serviti immediatamente, considerato che i due posti a disposizione sono liberi; dopodiché, i processi seguenti, Nino, Dino e Tino, troveranno il pronto soccorso occupato e verranno accodati e serviti in base all'ordine di arrivo e secondo il codice di gravità. Pertanto, tenendo conto di ciò, l'ordine totale con cui i pazienti verranno curati sarà: Pino, Gino, Tino (codice rosso), Dino (codice giallo), Nino.

`PreemptiveResource` Con la aggiunta delle risorse a priorità abbiamo indubbiamente ampliato la nostra capacità di modellazione, ma, sfortunatamente, ci sono ancora casi reali e frequenti che rimangono non rappresentabili. Infatti, sempre riferendoci all'esempio dell'ospedale, potremmo voler far sì che i pazienti con il codice rosso non solo abbiano precedenza nella coda, ma possano

addirittura "spodestare" altri pazienti già in cura, nel caso abbiano un codice meno grave. Questo concetto, denominato prelazione, è frequentemente utilizzato in ambito informatico [43] per consentire agli schedulatori di applicare alcuni meccanismi di *context switch* [44].

Pertanto, vista l'importanza di questo caso d'uso, gli sviluppatori di SimPy hanno creato un ulteriore raffinamento della classe `PriorityResource`, dando origine alla classe `PreemptiveResource`. Essa aggiunge ancora un parametro al metodo `request`, cioè, un booleano che indichi se il processo in questione abbia diritto di "scavalcare" gli utenti della risorsa con priorità minore. I processi scavalcati ricevono un interrupt e perdono definitivamente l'uso della risorsa; nel caso vogliano continuare a usarla, dovranno avanzare una nuova richiesta. L'istanza dell'interrupt contiene l'informazione su quale processo abbia fatto uso della prelazione per scavalcare gli altri utenti.

Alla luce di quanto descritto, abbiamo modificato l'esempio del pronto soccorso, facendo sì che i pazienti con il codice rosso avessero il diritto di prendere il posto degli altri pazienti meno gravi. I cambiamenti più sostanziali, presenti nel listato 2.22, consistono nell'aggiunta del parametro di prelazione al metodo `request` e l'uso del costrutto `try-except` per gestire la ricezione di eventuali interruzioni.

Come riportato nel listato 2.23, abbiamo che Pino (codice giallo) e Gino (codice verde) sono in cura, quando arriva Tino (codice rosso), il quale, in base a quanto abbiamo detto, ha diritto al posto di uno dei due. Tra Pino e Gino, viene scelto il secondo, dato che è quello avente priorità minore. Dopodiché, tutto procede come nell'esempio in 2.21, poiché non abbiamo inserito altri codici rossi all'interno della simulazione.

Attributi comuni Ciascun tipo di risorsa offre la possibilità di leggere la capacità massima e il numero attuale di utenti, rispettivamente tramite le proprietà `capacity` e `count`, e permette di vedere quali processi siano in coda, con la proprietà `queue`, e quali stiano usufruendo della risorsa stessa, tramite `users`.

2.5.2 `Store` e `FilterStore`

La simulazione a eventi discreti viene spesso usata in ambito manifatturiero [18, 19, 20], in quanto risulta utile per analizzare l'efficienza dei processi aziendali e consente di studiare eventuali approcci per migliorarli; all'interno di [18] viene proprio usata la libreria SimPy per aumentare l'efficienza di una forza lavoro data. In ambito aziendale, in particolare nei processi di produzione e di logistica, è necessario interagire con dei *magazzini*, nei quali possono essere conservati le materie prime o i prodotti finiti; oppure, sempre in quei processi, occorre trattare dei punti di *interscambio*, nei quali una parte della produzione sistema le proprie lavorazioni in modo tale che la parte successiva le possa prendere per raffinarle ulteriormente.

Per rappresentare quel tipo di situazioni, abbiamo bisogno di introdurre una nuova risorsa, la classe `Store`. Essa ci permette di avere una risorsa condivisa, nella quale i processi possono immagazzinare unità di oggetti, tramite il metodo

Listato 2.22 Diversa gestione dei codici rossi in un pronto soccorso.

```
from simpy import *

# Priorita' decrescenti, piu' una priorita' e' bassa
# prima il processo sara' posizionato nella coda.
RED, YELLOW, GREEN = 0, 1, 2

def person(env, name, code, hospital, delay):
    yield env.timeout(delay)
    env.active_process.name = name
    preempt = code == RED
    with hospital.request(code, preempt) as req:
        yield req
        print("%s viene curato..." % name)
        try:
            yield env.timeout(7)
            print("Cure finite per %s" % name)
        except Interrupt, i:
            byName = i.cause.by.name
            print("%s scavalcato da %s" % (name, byName))

env = Environment()
hospital = PreemptiveResource(env, capacity=2)
env.start(person(env, "Pino", YELLOW, hospital, 0))
env.start(person(env, "Gino", GREEN, hospital, 0))
env.start(person(env, "Nino", GREEN, hospital, 1))
env.start(person(env, "Dino", YELLOW, hospital, 1))
env.start(person(env, "Tino", RED, hospital, 2))
env.run()
```

Listato 2.23 Possibile output dello script in 2.22.

```
Pino viene curato...
Gino viene curato...
Gino scavalcato da Tino
Tino viene curato...
Cure finite per Pino
Dino viene curato...
Cure finite per Tino
Nino viene curato...
Cure finite per Dino
Cure finite per Nino
```

`put`, e possono recuperarli tramite il metodo `get`; entrambi i metodi possono essere utilizzati in modo *bloccante*, dato che restituiscono un evento che si attiva quando l'oggetto è stato realmente immagazzinato o recuperato. Infatti, occorre tenere conto del fatto che gli *store* possono avere una capacità prefissata; per cui, nel caso la quantità immagazzinata la eguagli, le operazioni di `put` saranno bloccate fino a che il magazzino non si svuoterà.

Analogamente, l'operazione di `get` sarà bloccante nel momento in cui si cercherà di estrarre un oggetto da un magazzino vuoto.

Considerato che questa situazione può essere ritenuta comune, approfittiamo dell'occasione per illustrare un uso molto pratico degli eventi condizione. In particolare, supponiamo di avere un processo che cerchi di prendere un oggetto da uno store, ma che non sia disposto ad aspettare in eterno che qualcuno immagazzini quanto richiesto. In tal caso, una prassi piuttosto comune è mettere l'evento restituito dalla `get` in disgiunzione con un timeout prestabilito; così, se l'operazione di recupero richiede troppo tempo, il processo sarà svegliato dal timeout e potrà prendere le opportune decisioni. Inoltre, ricordiamo che nel valore restituito dagli eventi condizione sono presenti tutti e soli gli eventi che sono stati attivati *prima* dell'attivazione complessiva della condizione.

Va anche sottolineato il fatto che tutte e tre le code presenti in uno store, cioè:

1. I processi che vorrebbero immagazzinare un oggetto.
2. I processi che vorrebbero prelevare un oggetto.
3. Gli oggetti che sono conservati.

Sono gestite con una politica *first in, first out*; pertanto, l'ordine con cui gli oggetti saranno recuperati dipenderà unicamente dall'ordine con cui sono stati inseriti.

Come consueto, chiudiamo la parte relativa alla classe `Store` riportando, nel listato 2.24, il classico esempio del produttore-consumatore [45] implementato su SimPy. Nel dettaglio, abbiamo che due processi `producer` memorizzano numeri interi casuali su uno store, dal quale il processo `consumer` li estrae e li processa. Considerato che il magazzino ha solamente capacità due, i processi produttori sfruttano gli eventi restituiti dalle operazioni `put` per sapere quando bloccarsi in attesa che il consumatore abbia liberato sufficiente spazio. Nel listato 2.25 riportiamo un possibile output dello script di esempio.

La classe `FilterStore` aggiunge delle funzionalità di filtraggio al metodo `get` di `Store`; in pratica, è possibile specificare, tramite una funzione, quali oggetti abbiano le caratteristiche richieste, in modo che il processo riceva *non* il primo oggetto disponibile, ma il primo oggetto *corretto* disponibile. Da questo punto di vista, un normale `Store` può essere pensato come un `FilterStore` dove il predicato di selezione restituisca sempre vero. Quindi, possiamo affermare che il comportamento di `FilterStore` è equivalente a `Store`, se non che i consumatori non sono svegliati appena un *qualunque* oggetto è disponibile, ma appena lo è uno che passa il loro filtro.

Listato 2.24 Classico esempio del produttore-consumatore.

```
1  from random import *
2  from simpy import *
3
4  rand = Random(21)
5  def randInt(): return rand.randint(1, 20)
6
7  def producer(env, store):
8      while True:
9          yield env.timeout(randInt())
10         item = randInt()
11         yield store.put(item)
12         print("%d: Prodotto un %d" % (env.now, item))
13
14 def consumer(env, store):
15     while True:
16         yield env.timeout(randInt())
17         item = yield store.get()
18         print("%d: Consumato un %d" % (env.now, item))
19
20 env = Environment()
21 store = Store(env, capacity=2)
22 env.start(producer(env, store))
23 env.start(producer(env, store))
24 env.start(consumer(env, store))
25 env.run(until=60)
```

Listato 2.25 Possibile output dello script in 2.24.

```
1  14: Prodotto un 1
2  14: Consumato un 1
3  20: Prodotto un 7
4  28: Prodotto un 12
5  33: Consumato un 7
6  33: Prodotto un 16
7  51: Consumato un 12
8  51: Prodotto un 8
9  54: Consumato un 16
10 54: Prodotto un 14
11 55: Consumato un 8
12 59: Prodotto un 13
```

Listato 2.26 Aggiunta di filtri all'esempio del produttore-consumatore.

```
1  from random import *
2  from simpy import *
3
4  rand = Random(63)
5  def randInt(): return rand.randint(1, 20)
6
7  def producer(env, store):
8      while True:
9          yield env.timeout(randInt())
10         item = randInt()
11         yield store.put(item)
12         print("%d: Prodotto un %d" % (env.now, item))
13
14 def consumer(env, store, name, filter):
15     while True:
16         yield env.timeout(randInt())
17         i = yield store.get(filter)
18         print("%d: %s, consumato un %d" % (env.now, name, i))
19
20 env = Environment()
21 store = FilterStore(env, capacity=2)
22 env.start(producer(env, store))
23 env.start(producer(env, store))
24 env.start(consumer(env, store, "PARI", lambda i: i%2 == 0))
25 env.start(consumer(env, store, "DISPARI", lambda i: i%2 == 1))
26 env.run(until=40)
```

A causa di ciò, mentre le code di inserimento e gli oggetti stessi sono sempre gestiti con politiche *first in, first out*, abbiamo che le code di prelievo sono *generalmente* gestite in quel modo, ma non sempre. Se il primo elemento della coda di prelievo non accetta il nuovo oggetto a disposizione, esso verrà passato al successore. Nel caso in cui esso lo accetti, allora il successore uscirà dalla coda *prima* del predecessore, il che viola le politiche *FIFO*.

Per il resto, non vi sono ulteriori funzionalità aggiuntive. Dunque, il listato 2.26 contiene una versione modificata dell'esempio in 2.24, dove vengono aggiunti dei filtri ai consumatori, in modo che uno legga solo i numeri pari e l'altro solo quelli dispari. La funzione di filtraggio, specificata come *lambda*, viene usata a ogni invocazione dell'operazione di `get`. Un possibile output dello script è mostrato nel listato 2.27.

Attributi comuni Gli attributi rilevanti presenti in entrambi i tipi di store sono `capacity`, dal quale si recupera la capacità assegnata, e `items`, con il quale

Listato 2.27 Possibile output dello script in 2.26.

```
7: Prodotto un 13
7: DISPARI, consumato un 13
13: Prodotto un 1
13: DISPARI, consumato un 1
20: Prodotto un 15
24: DISPARI, consumato un 15
29: Prodotto un 5
31: Prodotto un 8
31: PARI, consumato un 8
34: Prodotto un 8
34: PARI, consumato un 8
38: Prodotto un 7
```

è possibile sapere quali elementi sono presenti nel magazzino.

2.5.3 Container

Se il lettore ha compreso il funzionamento degli store, allora non avrà problemi a capire l'uso della classe `Container` e, forse, ne avrà già intuito la ragion d'essere. Rimanendo sempre in ambito manifatturiero, abbiamo che i magazzini non sono i soli luoghi dedicati alla conservazione dei materiali, o meglio, essi possono solo serbare unità ben definite di materiale, come una scatola o degli oggetti. Tuttavia, come modellare depositi di acqua, benzina o sabbia, dove non sono le singole unità a essere rilevanti, ma lo è il volume totale? Bene, i container sono la risposta di SimPy a quel quesito. Essi hanno un'interfaccia del tutto analoga agli store, ma il loro scopo è memorizzare quantità di un certo materiale, per poi consentire, con gli analoghi metodi `get` e `put`, di aggiungerne o prelevarne frazioni arbitrarie.

Per aiutare il lettore a comprendere meglio la differenza, immaginiamo di essere in una cucina e dover aggiungere il sale a una pietanza. I pacchi di sale, in genere da un chilogrammo, saranno conservati nella dispensa, mentre il sale stesso, una volta estratto dal pacco, sarà contenuto nella saliera. Dovendo rapportare quanto detto al *mondo* di SimPy, possiamo associare uno store di pacchetti di sale alla dispensa, mentre la saliera sarà un container di sale con la capacità attorno al chilogrammo. Dalla dispensa preleviamo unità di sale, i pacchetti, mentre dalla saliera ne prendiamo dei volumi, come i cucchiai o le manciate.

Il listato 2.26, e il relativo output in 2.27, contiene un possibile uso dei container, che vengono usati per modellare un distributore di bibite. Il distributore è un container avente capacità da un litro, dal quale le persone prelevano un quarto di litro per riempire i propri bicchieri; quando una persona si accorge che il distributore è vuoto, chiama un tecnico in modo che lo riempia. Si noti

come i metodi `get` e `put` richiedano che la quantità in questione sia specificata come parametro.

Attributi La classe `Container` espone un attributo `capacity`, per vedere la capacità assegnata alla risorsa stessa, e un attributo `level`, per poter conoscere il volume attualmente contenuto.

2.6 Esempio della banca

In sezione 1.2.2 abbiamo introdotto un semplice esempio, con il quale abbiamo cercato di spiegare quali fossero i principali approcci alla costruzione di un simulatore. A questo punto, dopo aver visto gli elementi essenziali della libreria SimPy, faremo il passo successivo, cioè, proveremo a implementare tale modello, per mostrare come sia possibile combinare le componenti che abbiamo descritto finora e per mettere in luce quali siano i risvolti pratici della simulazione stessa.

Pertanto, riproporremo il modello, presentato in sezione 1.2.2, con alcune lievi modifiche che ci permetteranno di fare osservazioni più specifiche; dopodiché, creeremo passo passo le componenti necessarie per il nostro sistema.

Il codice completo dell'esempio è riportato in appendice A.

2.6.1 Modello della simulazione

Per prima cosa, partiamo dai dati a nostra disposizione. Supponiamo di avere che:

- I clienti arrivino circa ogni tre minuti e che i servizi da loro richiesti consumino dieci minuti in media.
- Vogliamo analizzare la situazione in banca nell'arco di un turno di lavoro, cioè, cinque ore.
- La banca ha un deposito di denaro che, per questioni di sicurezza, non può ospitare più di ventimila euro.
- La banca apre con duemila euro contanti.
- Ciascun cliente può prelevare o depositare una cifra che varia dai 50 ai 500 euro; se la banca non ha sufficiente contante, si dovrà aspettare che qualche altro cliente depositi una cifra sufficiente.
- Mediamente, il 60% dei clienti depositerà, mentre il restante 40% preleverà.
- La nostra sede bancaria ha tre sportelli aperti, per tutto il turno.

Detto ciò, un cliente farà le seguenti operazioni, nell'ordine indicato:

1. Ricerca della coda più corta e sistemazione in essa.

Listato 2.28 Esempio d'uso della classe `Container`.

```
1  from simpy import *
2
3  boxCapacity = 1 # Litri
4  glassCapacity = 0.25 # Litri
5  fillBox = None
6
7  def filler(env, box):
8      global fillBox
9      while True:
10         yield box.put(boxCapacity - box.level)
11         fillBox = env.event()
12         id = yield fillBox
13         print("%d: %d chiama tecnico" % (env.now, id))
14
15 def drinker(env, id, box):
16     # Occorre controllare che l'evento fillBox non sia gia'
17     # stato attivato, perche' attivarlo nuovamente
18     # risulterebbe in una eccezione da parte di SimPy.
19     if box.level < glassCapacity and not fillBox.triggered:
20         fillBox.succeed(id)
21     yield box.get(glassCapacity)
22     print("%d: %d ha bevuto!" % (env.now, id))
23
24 def spawner(env, box):
25     id = 0
26     while True:
27         yield env.timeout(5)
28         env.start(drinker(env, id, box))
29         id += 1
30
31 env = Environment()
32 box = Container(env, capacity=boxCapacity)
33 env.start(filler(env, box))
34 env.start(spawner(env, box))
35 env.run(until=31)
```

Listato 2.29 Possibile output dello script in 2.28.

```
5: 0 ha bevuto!
10: 1 ha bevuto!
15: 2 ha bevuto!
20: 3 ha bevuto!
25: 4 chiama tecnico
25: 4 ha bevuto!
30: 5 ha bevuto!
```

2. Attesa del proprio turno.
3. Una volta allo sportello, effettuerà il deposito o il prelievo.
4. Fatto quanto dovuto, il cliente uscirà dalla banca.

Infine, vogliamo tenere traccia di queste variabili di interesse:

- Il tempo che un cliente, in media, passa in coda.
- Il tempo medio impiegato per servire i clienti.
- La percentuale di clienti servita.
- I contanti con cui la banca chiude il turno.

2.6.2 Variabili richieste e accessorie

Il listato 2.30 contiene le dichiarazioni preliminari necessarie per il buon funzionamento della simulazione. Notiamo che, per prima cosa, viene dichiarato il generatore di numeri casuali, il quale verrà utilizzato per stabilire casualmente i tempi di arrivo dei clienti, il loro tempo di servizio e il fatto che siano venuti per un prelievo o un deposito.

Successivamente, dichiariamo alcune costanti relative ai dati del nostro modello; ciò ha lo scopo di rendere più leggibile il codice e ci consentirà di "sperimentare" con la simulazione, semplicemente variando i dati iniziali. Poi inizializziamo le variabili di accumulazione che useremo per raccogliere dati statistici e, infine, allochiamo le risorse di SimPy necessarie. In particolare, abbiamo bisogno di un ambiente di esecuzione, di tre risorse per simulare gli sportelli e di un container per modellare il nostro deposito monetario.

Le risorse, a capacità uno, offrono in automatico i meccanismi di accodamento dei clienti, mentre il container permette di bloccare quei clienti che vorrebbero prelevare più di quanto la banca possiede o versare una somma che farebbe superare alla banca il tetto massimo.

Listato 2.30 Parte iniziale della nostra simulazione di una banca.

```
1 from random import *
2 from simpy import *
3
4 rand = Random(21)
5 exp = lambda x: rand.expovariate(1.0/x)
6
7 avgIncomingT, avgServiceT = 3.0, 10.0 # Minuti
8 queueCount = 3 # Numero sportelli
9 bankCap, bankLvl = 20000, 2000 # Euro
10
11 # Usate per raccogliere dati statistici
12 totClients, totServedClients = 0, 0
13 totWaitT, totServT = 0.0, 0.0
14
15 env = Environment()
16 queues = [Resource(env, 1) for i in range(queueCount)]
17 bank = Container(env, bankCap, bankLvl)
```

2.6.3 Il processo cliente

Il processo che modella la "vita" di un cliente, riportato nel listato 2.31, dovrebbe risultare abbastanza chiaro al lettore. Ricapitolando, abbiamo preso le specifiche indicate in sezione 2.6.1 e le abbiamo tradotte, un passo alla volta, nel linguaggio SimPy; quindi: un cliente inizia con il mettersi nella coda più corta, selezionata e passata come parametro da `spawner` (mostrato nel listato 2.32 e descritto nella prossima sezione) e si arresta per aspettare il proprio turno. Una volta allo sportello, egli aspetta un tempo casuale per simulare lo svolgimento delle pratiche. Infine, in base al compito che gli è stato assegnato, prova a prelevare o a depositare il denaro richiesto; nel caso in cui il deposito non ne contenga una quantità sufficiente, aspetterà che qualcuno faccia depositi a sufficienza.

Il corpo del processo contiene anche le istruzioni necessarie per registrare il tempo di attesa e di servizio, modificando opportunamente le variabili globali adibite allo scopo.

2.6.4 Il processo generatore di clienti

I clienti vengono generati da un processo apposito, presente nel listato 2.32, in modo tale che ne arrivi uno ogni tre minuti. Inoltre, è compito di questo processo scegliere la coda di minore lunghezza, in modo che il cliente sappia dove accordarsi. Ai fini della raccolta di dati statistici, il generatore tiene traccia di quanti nuovi clienti siano stati immessi nella banca.

Listato 2.31 Definizione del processo relativo ai clienti.

```
def client(env, queue, bank, amount, get):
    global totServedClients, totWaitT, totServT
    with queue.request() as req:
        start = env.now
        yield req
        totWaitT += env.now - start
        start = env.now
        yield env.timeout(exp(avgServiceT))
        if get: yield bank.get(amount)
        else: yield bank.put(amount)
        totServT += env.now - start
        totServedClients += 1
```

Listato 2.32 Definizione del processo generatore di clienti.

```
def spawner(env, queues, bank):
    global totClients
    while True:
        yield env.timeout(exp(avgIncomingT))
        queue = min(queues, key=lambda q:len(q.queue))
        amount = rand.uniform(50, 500)
        get = rand.random() < 0.4
        env.start(client(env, queue, bank, amount, get))
        totClients += 1
```

Listato 2.33 Avvio della simulazione e conseguente raccolta dati.

```
# Avvio della simulazione
env.start(spawner(env, queues, bank))
env.run(until=5*60)

# Raccolta dati statistici
lvl = bank.level
print("Finanze totali al tempo %.2f: %d" % (env.now, lvl))
print("Clienti entrati: %d" % totClients)
print("Clienti serviti: %d" % totServedClients)
avgWait = totWaitT/totServedClients
print("Tempo medio di attesa: %.2f" % avgWait)
avgServ = totServT/totServedClients
print("Tempo medio di servizio: %.2f" % avgServ)
```

2.6.5 Avvio della simulazione e raccolta dati

La parte di gestione della simulazione e della raccolta dei dati statistici è delegata al codice riportato nel listato 2.33. In pratica, ciò che si fa è semplicemente avviare il processo generatore di clienti, al fine di dargli la possibilità di popolare la banca; dopodiché, si avvia la simulazione e si chiede di farla durare per trecento minuti, cioè, cinque ore.

Per ottenere i tempi medi di attesa e di servizio, ciò che facciamo è semplicemente prendere i totali e dividerli per il numero di clienti serviti. Invece, la quantità di contanti rimasti in banca è ottenuta tramite la proprietà `level` del container a essa relativo.

2.6.6 Analisi dei dati

Una volta che la nostra simulazione è pronta, la possiamo finalmente eseguire. Un possibile output, ottenuto unendo il codice riportato nei listati 2.30, 2.31, 2.32 e 2.33, risulta essere:

```
Finanze totali al tempo 300.00: 6118
Clienti entrati: 99
Clienti serviti: 84
Tempo medio di attesa: 17.66
Tempo medio di servizio: 9.79
```

Notiamo immediatamente che molti clienti non sono stati serviti e che i tempi di attesa sono molti lunghi. Qui entra in gioco la vera utilità della simulazione: come possiamo ridurre i tempi di attesa e aumentare il numero di clienti serviti? Mettendoci nei panni del "gestore" della banca, possiamo lavorare su tre parametri:

1. Il numero di sportelli.

2. Il numero massimo di contanti conservabili nella banca.

3. Il numero di contanti disponibili al mattino.

Provando, solo per curiosità, a raddoppiare le quantità indicate ai punti due e tre, otteniamo i seguenti risultati:

```
Finanze totali al tempo 300.00: 8118
Clienti entrati: 99
Clienti serviti: 84
Tempo medio di attesa: 17.66
Tempo medio di servizio: 9.79
```

Come ci si sarebbe potuti aspettare, nulla è cambiato, se non la quantità finale di contanti. Se facciamo la cosa più sensata, cioè, raddoppiamo il numero di sportelli, otteniamo:

```
Finanze totali al tempo 300.00: 6571
Clienti entrati: 105
Clienti serviti: 103
Tempo medio di attesa: 5.72
Tempo medio di servizio: 9.59
```

Il risultato è notevole, in quanto abbiamo incredibilmente aumentato il numero di clienti serviti e abbiamo ridotto di ben dodici minuti i tempi medi di attesa. Arrivati a ciò, in un caso reale, sarebbe inopportuno fermare le analisi: infatti, raddoppiare gli sportelli avrebbe un costo economico notevole, il che ci dovrebbe spingere a cercare un'ottimizzazione intermedia tra l'avere bassi tempi di attesa, ma grandi costi in termini di personale, e lunghi tempi di attesa e un personale più snello.

Il succo della simulazione è proprio questo: facilitare l'analisi e l'ottimizzazione di processi che, da un punto di vista strettamente analitico, sarebbero di difficile rappresentazione. Il paradigma a processi, seguito fedelmente da SimPy, ci consente di mappare il mondo reale con il mondo virtuale delle simulazioni, evitandoci l'introduzione di grandi astrazioni e rendendo l'analisi del modello un'operazione più intuitiva.

Capitolo 3

Prerequisiti di Dessert

L'obiettivo della tesi, come indicato in sezione 1.1, era replicare le funzionalità di SimPy, ampiamente descritte nel capitolo 2, su un linguaggio fortemente tipato e, soprattutto, compilato, al fine di ottenere prestazioni e robustezza del codice decisamente superiori a quelle della libreria Python. In ogni caso, il nostro *clone* avrebbe dovuto rispettare i principi chiave che stanno dietro al design di SimPy che abbiamo maggiormente apprezzato; ciò implicava la scelta di un linguaggio che offrisse costrutti analoghi alle coroutine e con il quale fosse possibile offrire un'interfaccia simile a quella di SimPy.

Mentre il secondo vincolo non ha creato alcun tipo di problema, un qualunque linguaggio orientato agli oggetti sarebbe stato sufficiente, il primo vincolo ci ha costretto a scremare la maggior parte dei linguaggi a disposizione, al punto da lasciarci un numero esiguo di alternative. I dettagli saranno raccontati in sezione 3.1, dove si scoprirà che la scelta è stata praticamente obbligata.

Una volta stabilito che avremmo lavorato sulla piattaforma .NET, ci siamo chiesti quali fossero gli *ingredienti* necessari per poter realizzare un'efficiente e corretta implementazione di un motore per la simulazione a eventi discreti. In primis, ci serviva un'efficiente coda a priorità, con la quale gestire l'event set del simulatore stesso; poi, occorreva un generatore di numeri casuali secondo certe distribuzioni di probabilità (normale ed esponenziale, ad esempio), strettamente necessario nell'ambito della modellazione della realtà ai fini della simulazione.

Sfortunatamente, né le code a priorità né i generatori di numeri casuali erano presenti nella libreria standard di .NET: l'unico rimedio a tale problema è stato cercare l'esistenza di eventuali alternative o, nel caso le alternative non ci soddisfacessero o non fossero presenti, scrivere direttamente noi il codice necessario. Piuttosto che aggiungere le strutture mancanti al codice del simulatore stesso, abbiamo scelto di creare tante librerie separate. Ciò non è stato fatto soltanto con lo scopo di ottenere un buon design, ma ha anche per consentire la pubblicazione delle librerie stesse sulla piattaforma NuGet [46], dove possono essere consumate facilmente dagli utenti degli ambienti di sviluppo Visual Studio [47] e SharpDevelop [48] (e in futuro, forse, anche MonoDevelop [49, 50]).

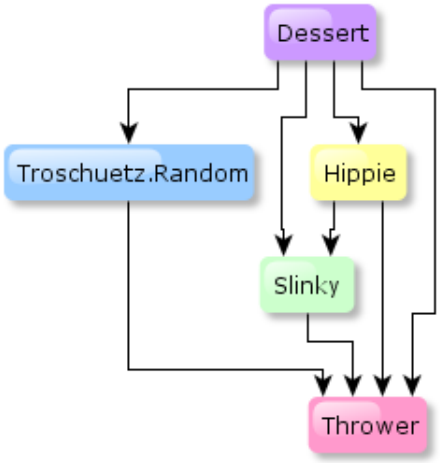


Figura 3.1: Grafo delle dipendenze per le librerie realizzate. La freccia indica la libreria dalla quale si dipende.

In merito alle code a priorità, abbiamo scelto di scrivere una libreria a parte, *Hippie* [51, 52], che verrà dettagliata in sezione 3.2; per quanto riguarda la generazione di numeri casuali, abbiamo preso i sorgenti di una vecchia libreria rinvenuta su internet, *Troschuetz.Random* [53, 54], li abbiamo migliorati e resi più moderni, come descritto in sezione 3.3.

Nel corso dello sviluppo ci siamo imbattuti in un'altra questione: l'efficiente scrittura dei controlli di integrità. In breve, ci siamo posti la domanda se ci fosse un modo per avere dei controlli di integrità *disabilitabili*, in modo da poterli sopprimere nelle situazioni dove le massime prestazioni sono richieste. Da tale dubbio è nata la libreria *Thrower* [55, 56], descritta in sezione 3.4, con la quale abbiamo risolto la questione.

Sempre spinti dalla necessita di avere un simulatore decisamente performante, abbiamo scritto un'ulteriore libreria, *Slinky* [57, 58], descritta in sezione 3.5, nella quale implementiamo in modo semplice ed efficiente diversi tipi di liste linkate.

Tutte e quattro le librerie sono *open source* e, per la precisione, i sorgenti sono rilasciati secondo la licenza *MIT/X11*[1] [60].

Come mostrato in figura 3.1, non solo Dessert si serve delle librerie create, ma sono presenti altre relazioni di dipendenza; ad esempio, in tale figura si può osservare che Hippie usa sia Slinky sia Thrower, mentre Thrower è usata da tutte le altre librerie. Questo ci sembra un buon risultato a livello di riuso del codice, in quanto abbiamo creato un piccolo "ecosistema" di librerie che si supportano a vicenda, pur mantenendo interfacce il più generiche possibile.

[1] Fatta eccezione per la libreria Troschuetz.Random, per la quale abbiamo mantenuto la licenza originale (*LGPLv2.1* [59]).

Proseguiamo il capitolo descrivendo, in sezione 3.1, le nostre indagini per la scelta della piattaforma. Successivamente, nelle sezioni da 3.2 a 3.5, descriveremo le librerie esterne in modo più estensivo, al fine di rendere più chiare sia le scelte fatte, sia le funzionalità esposte; tuttavia, non si scenderà troppo nel dettaglio, in quanto lo scopo continua a essere quello di fare una panoramica dei prerequisiti di Dessert.

3.1 Panoramica delle coroutine

Secondo quanto riportato in [2], in molti linguaggi esiste un supporto nativo per le coroutine; tuttavia, tra quelli, solo pochi sono fortemente tipati e compilati, come richiesto dai nostri requisiti. In particolare, tra i linguaggi fortemente tipati possiamo annoverare D, Erlang, Haskell, Go, C#, VB.NET e F#; tuttavia, gli unici a noi noti erano quelli su .NET e imparare un nuovo linguaggio avrebbe reso il progetto ancora più arduo di quanto già non fosse.

Però, sempre secondo [2], per altri linguaggi esiste la possibilità di aggiungere il supporto alle coroutine, il che ci ha permesso di includere nella valutazione C, C++ e Java. Quindi, vediamo ora perché fra le alternative abbiamo optato proprio per la piattaforma .NET.

3.1.1 Disponibilità in C e in C++

Come si sa, il C e il C++ non offrono supporto per le coroutine e solo il C++ ha un'interfaccia standard, gli *iteratori* [61], per rappresentare sequenze potenzialmente infinite di oggetti. Nonostante ciò, esistono alcuni metodi per aggiungere il supporto alle coroutine; tali metodi generalmente si servono delle *macro* per riuscire a manipolare il codice sorgente prima che venga compilato [62] oppure, solo in C++, sfruttano la metaprogrammazione e/o la manipolazione dello stack [63, 64].

Quei metodi, però, hanno alcuni svantaggi diretti e indiretti:

- Il metodo in [62] richiede che le variabili di un metodo siano dichiarate come `static`; nel nostro caso un vincolo di quel tipo sarebbe troppo stringente, dato che le coroutine corrisponderebbero con i processi e ci sarebbero sicuramente più invocazioni della stessa coroutine in una singola simulazione.

- Il metodo in [63] sembra ben funzionante, ma usare tale libreria implicherebbe che gli utenti del nostro progetto debbano essere esperti in C++. Infatti, la libreria sembra piuttosto complicata da usare, e il codice risultante è decisamente molto meno elegante rispetto a quello Python.

- Il metodo in [64] soffre degli stessi difetti di quello in [63] ed è, in ogni caso, solamente una proposta. Infatti, nel documento si può leggere che, per il buon funzionamento della libreria, è richiesto che il compilatore sappia

gestire *stack segmentati*, ma attualmente solo GCC[2], versione 4.7, ne è capace [66]. Ad ogni modo, limitare la scelta dei nostri utenti a un solo compilatore non sembrava una buona mossa.

Pertanto, sia per questioni estetiche sia per ragioni tecniche, abbiamo messo da parte C e C++.

3.1.2 Disponibilità in Java

Di per sè, la piattaforma Java non ha un supporto nativo per le coroutine, ma esistono tutta una serie di librerie che o le emulano tramite uso dei thread [67] o le implementano tramite weaving [68] del bytecode per la JVM [69, 70, 71, 72, 73].

Le soluzioni basate su thread devono necessariamente usare un thread per ogni coroutine, il che le ha rese assolutamente inadatte per il nostro progetto. Potenzialmente, una simulazione potrebbe anche coinvolgere milioni di processi, e ciò implicherebbe avere altrettanti thread attivi: non solo il consumo in memoria sarebbe elevato, ma molti cicli di clock andrebbero persi soltanto per gestire l'enorme quantità di thread allocati.

Invece, le soluzioni basate su weaving sarebbero state ottime, non fosse che *nessuna* ha un supporto stabile; infatti, tutte sono piuttosto datate e sono frutto degli sforzi di individui che hanno dedicato parte del loro tempo libero a quei progetti. Considerata la complessità di gestire correttamente la trasformazione del codice in modo che diventi una coroutine [74], abbiamo ritenuto poco affidabile quel codice e abbiamo deciso di non prenderci carico della gestione e della modifica di una di quelle librerie, perché il carico di lavoro avrebbe rischiato di sorpassare quello necessario per il simulatore.

3.1.3 Disponibilità su .NET

I principali linguaggi per la piattaforma .NET (C#, VB.NET[3], F#) sono tra i pochi "fortunati" ad avere le coroutine integrate direttamente dentro il linguaggio. Le implementazioni per C# e VB.NET sfruttano il design pattern degli iteratori per ottenere, tramite una trasformazione a tempo di compilazione, il risultato voluto; invece, F# segue un'altra strada [76], pur ottenendo le medesime conclusioni.

Nei listati 3.1, 3.2 e 3.3 mostriamo un uso molto semplice delle coroutine rispettivamente in C#, VB.NET e F#.

Mentre in Python una coroutine può ricevere e restituire un valore a ogni iterazione, in quei linguaggi per .NET i valori possono soltanto uscire da una coroutine. Sfortunatamente, i generatori bidirezionali sono usati da SimPy, il che ci ha costretto, per arginare il problema, a fare delle opportune scelte di design nella nostra implementazione.

[2]GNU Compiler Collection [65] è una famosa e importante interfaccia per una serie di compilatori di linguaggi, tra cui abbiamo quelli per il C, C++, Go, Fortran e Ada.

[3]Solo dalla versione 11 del linguaggio [75].

Listato 3.1 Generatore di numeri di Fibonacci in C#.

```
IEnumerable<int> Fibonacci() {
    int a = 1, b = 0;
    while (true) {
        yield return a;
        b = a + b;
        yield return b;
        a = a + b;
    }
}
```

Listato 3.2 Generatore di numeri di Fibonacci in VB.NET.

```
Iterator Function Fibonacci() As IEnumerable(Of Integer)
    Dim a = 1
    Dim b = 0
    While True
        Yield a
        b = a + b
        Yield b
        a = a + b
    End While
End Function
```

Listato 3.3 Generatore di numeri di Fibonacci in F#.

```
let rec auxFib a b = seq {
    yield a
    let c = a + b
    yield c
    yield! auxFib (a+c) c
}

let fibonacci = auxFib 1 0
```

Come si è visto negli esempi presentati in precedenza, l'uso delle coroutine in .NET è pressoché tanto elegante quanto quello di Python; ciò è stato un forte punto a favore di questa scelta. Inoltre, l'aver optato per tale piattaforma ci ha consentito di integrare i servizi di IronPython [77] nella nostra libreria, in modo tale da poter sperimentare con Armando, descritto nel capitolo 4.

3.2 Libreria Hippie

Curiosamente, la libreria standard dell'ambiente .NET non contiene alcuna implementazione della struttura dati *heap*, necessaria per realizzare un motore per la simulazione a eventi discreti [78]; la mancanza risulta strana poiché le librerie standard di altri linguaggi molto diffusi, come Java e C++, ne possiedono un'implementazione [79, 80]. Pertanto, al fine di colmare tale lacuna, è stata creata la libreria *Hippie*, la quale implementa quattro tipi di heap; uno basato su array [81], come il classico heap binario, e tre (binomiale [82], Fibonacci [83], pairing [84]) basati sulla struttura *esplicita*[4] ad albero.

Nonostante implementare un solo tipo di heap sarebbe stato sufficiente per il nostro scopo, si è scelto di sperimentare con diversi tipi di strutture non studiate durante i corsi, con la speranza di trovarne qualcuna in grado di fornire prestazioni eccellenti all'interno di Dessert. Nel capitolo 5 verranno effettuati dei confronti dettagliati tra le prestazioni offerte da ciascun tipo di heap in diversi casi d'uso del simulatore; tuttavia, è già possibile anticipare che la migliore struttura dati è risultata essere quella basata su array: la sua semplicità e il suo basso *overhead* le consentono di realizzare le operazioni più comuni con notevole velocità.

Originariamente, la libreria Hippie era stata realizzata come progetto finale per il corso di *Modular and Generic Programming*, tenuto dai professori Giovanni Lagorio [85] e Davide Ancona [86]. Successivamente alla consegna del progetto, la libreria è stata ulteriormente arricchita, testata e documentata, con lo scopo di renderla sufficientemente solida da poter essere alla base del motore per la simulazione e da poter essere pubblicata su internet con una certa confidenza.

3.2.1 Interfaccia

Descriveremo ora come la libreria esponga i propri servizi base verso gli utenti, e quali classi/interfacce essi abbiano a disposizione per ottenere istanze degli heap. Prima di presentare una sintesi del codice, riteniamo opportuno fare un elenco delle operazioni richieste per ciascuno heap, in modo da comprendere al meglio le scelte fatte. Le operazioni sono:

- Aggiunta e rimozione di un elemento.

[4] Con struttura *esplicita* si intende che la configurazione ad albero viene preservata anche nella rappresentazione in memoria. Ciò non avviene per lo heap basato su array, dove la rappresentazione in memoria non ricalca la reale struttura ad albero (più precisamente, ne è una sua codifica basata su determinate convenzioni).

- Rimozione dell'elemento *minimo*, dove l'ordinamento viene stabilito da un comparatore opzionalmente definibile dall'utente.
- Aggiornamento della priorità di un elemento; questa operazione è fondamentale ai fini del simulatore[5], in quanto gli eventi vengono programmati con una certa priorità che sicuramente cambierà nel corso della simulazione.
- Possibilità di determinare rapidamente se un elemento sia contenuto o meno nello heap.
- Capacità di unire due heap, in modo tale che lo heap risultante sia comunque uno heap valido. Questa operazione non era strettamente richiesta per realizzare il simulatore, ma rappresentava comunque un interessante settore di analisi.

La possibilità di cambiare dinamicamente la priorità di un dato oggetto, in quanto requisito fondamentale del progetto, ha fatto sì che la libreria Hippie sia stata progettata per ridurre al minimo il tempo di esecuzione di tale operazione. In particolare, il cambiamento di priorità consiste in due fasi distinte:

1. Ricerca dell'elemento all'interno dello heap; ciò, a seconda della struttura dati, può richiedere anche $O(n)$.
2. Modifica della struttura sottostante allo heap; in generale, il costo delle modifiche è $O(log(n))$.

Ridurre il tempo di esecuzione della seconda fase non è tecnicamente possibile, dato che è strettamente legato alla struttura dati con la quale si implementa lo heap; pertanto, l'unica fase sulla quale abbiamo potuto lavorare è stata la prima. Si è introdotto il concetto di *handle*, che corrisponde a un riferimento sicuro ai nodi dello heap stesso. Gli handle vengono restituiti dall'operazione di aggiunta, mentre le operazioni che richiedono la ricerca di un elemento, come il cambiamento di priorità e la rimozione, richiedono come parametro un handle invece che un elemento. Considerato che l'handle *punta* direttamente all'interno dello heap, ciascuna implementazione lo espone in modo tale da consentire l'esecuzione in tempo costante della ricerca.

Detto ciò, possiamo passare al codice. Nel listato 3.4 è abbozzata l'interfaccia `IRawHeap` con il relativo handle, mentre nel listato 3.5 si presenta l'interfaccia `IHeap`, che offre le stesse operazioni della prima senza esporre alcun handle. La ragione è che `IHeap` è stata pensata per togliere all'utente il fardello della gestione degli handle, cosa della quale si occupano le implementazioni di tale interfaccia, ad esempio mappando gli elementi e i relativi handle verso un dizionario. Da ciò si può comprendere perché la prima interfaccia abbia l'aggettivo *raw* all'interno del proprio nome: essa è più vicina all'implementazione, può fornire prestazioni migliori (a discapito della facilità d'uso).

[5] Recenti test durante lo sviluppo di Dessert hanno suggerito un possibile approccio alternativo, nel quale l'operazione di cambiamento della priorità non risulta più necessaria. In ogni caso, la tecnica è specifica al caso di Dessert e si può continuare a ritenere tale operazione necessaria per un simulatore generico.

Listato 3.4 Dichiarazione dell'interfaccia `IRawHeap`.

```
1  interface IHeapHandle<out V, out P> {
2      V Value { get; }
3      P Priority { get; }
4  }
5
6  interface IRawHeap<V, P>
7      : ICollection<IHeapHandle<V, P>>
8  {
9      IComparer<P> Comparer { get; }
10     int MaxCapacity { get; }
11     IHeapHandle<V, P> Min { get; }
12     P this[IHeapHandle<V, P> handle] { set; }
13
14     IHeapHandle<V, P> Add(V val, P pr);
15
16     void Merge<V2, P2>(IRawHeap<V2, P2> other)
17         where V2 : V where P2 : P;
18
19     IHeapHandle<V, P> RemoveMin();
20
21     P UpdatePriorityOf(IHeapHandle<V, P> handle,
22                        P newPr);
23
24     V UpdateValue(IHeapHandle<V, P> handle,
25                   V newVal);
26
27     IEnumerable<IReadOnlyTree<V, P>> ToReadOnlyForest();
28 }
```

Listato 3.5 Dichiarazione dell'interfaccia `IHeap`.

```
1  interface IHeap<V, P>
2      : ICollection<IHeapHandle<V, P>>
3  {
4      IComparer<P> Comparer { get; }
5      IEqualityComparer<V> EqualityComparer { get; }
6      int MaxCapacity { get; }
7      IHeapHandle<V, P> Min { get; }
8      P this[V val] { get; set; }
9
10     void Add(V val, P pr);
11
12     bool Contains(V val);
13
14     bool Contains(V val, P pr);
15
16     void Merge<V2, P2>(IHeap<V2, P2> other)
17         where V2 : V where P2 : P;
18
19     P PriorityOf(V val);
20
21     IHeapHandle<V, P> Remove(V val);
22
23     IHeapHandle<V, P> RemoveMin();
24
25     P Update(V val, V newVal, P newPr);
26
27     P UpdatePriorityOf(V val, P newPr);
28
29     void UpdateValue(V val, V newVal);
30
31     IEnumerable<IReadOnlyTree<V, P>> ToReadOnlyForest();
32 }
```

Si sarà notato, leggendo la dichiarazione delle interfacce, che gli elementi dello heap sono coppie *(valore, priorità)*; ciò è stato fatto per garantire la massima flessibilità, in quanto non è detto che un oggetto abbia la priorità inclusa tra i propri campi. Ad esempio, supponendo di voler utilizzare uno heap per archiviare dei messaggi (stringhe) associando loro una certa urgenza (intera), senza il supporto alle coppie *(valore, priorità)* da parte dello heap sarebbe necessario aggregare le stringhe e gli interi in un terzo oggetto, il quale diverrebbe il tipo degli elementi della coda a priorità. Tuttavia, vi sono casi dove gli oggetti hanno davvero la priorità al loro interno; pertanto, è stata definita un'opportuna interfaccia per gestire al meglio anche tali casi, in modo che l'utente possa interagire facilmente e in modo intuitivo con la libreria.

Le istanze dei vari tipi di heap implementati possono essere recuperate tramite la classe statica `HeapFactory`, il cui codice non viene riportato nella relazione in quanto piuttosto prolisso e di poco interesse. Chiaramente, ciascun tipo di implementazione ha i propri vantaggi e svantaggi: le tre strutture basate sugli alberi espliciti hanno migliori tempi teorici, soprattutto per l'operazione di unione, mentre la struttura basata sui vettori, pur essendo leggermente peggiore a livello teorico, si comporta egregiamente nella pratica. Per chi fosse interessato all'argomento, in [89] viene presentata una tabella dove si confrontano le varie complessità temporali delle operazioni principali.

Occorre sottolineare il fatto che, di per sè, gli heap implementati non sono *stabili*, cioè, l'ordine con cui viene estratta una serie di elementi aventi la stessa priorità non è detto che coincida con l'ordine con cui tali elementi sono stati inseriti. Tuttavia, abbiamo applicato un meccanismo, lo stesso utilizzato dalla libreria C++ *Boost*[6] [88], che consente di rendere stabili anche code a priorità che non lo sarebbero *naturalmente*. Brevemente, l'idea chiave consiste nell'aggiungere una sorta di *numero di revisione* a ciascuna priorità, in modo che elementi arrivati prima abbiano numeri di revisione più bassi; tale operazione può essere realizzata in modo trasparente, senza alcun intervento dell'utente. La stabilità è un fattore di particolare importanza nell'ambito della simulazione, dato che essa rende *deterministica* la scelta di eventi programmati per lo stesso istante di tempo; cosa che, altrimenti, dipenderebbe dalla particolare implementazione di heap.

Le istanze "stabili" della struttura dati possono essere recuperate dalla classe statica `StableHeapFactory`, analoga a `HeapFactory`; all'interno di Dessert vengono usate ben due istanze di heap, come vedremo nel seguito.

Prima di concludere la breve presentazione di Hippie, mostreremo un paio di esempi d'uso e parleremo di come essa si rapporti con le code a priorità in Java e con la libreria *C5* per .NET.

[6] Boost [87] è una famosa libreria per il linguaggio C++ che contiene tutta una serie di funzioni e strutture dati non presenti nella libreria standard. La libreria è scritta e mantenuta in modo molto competente, al punto che spesso tale libreria viene considerata tanto importante quanto quella standard.

Listato 3.6 Implementazione *naïve* dell'algoritmo di heap sort.

```
1  IList<T> HeapSort<T>(IEnumerable<T> elems)
2      where T : IComparable<T>
3  {
4      var heap = HeapFactory.NewBinaryHeap<T>();
5      foreach (var elem in elems)
6          heap.Add(elem);
7      var list = new List<T>(heap.Count);
8      while (heap.Count != 0)
9          list.Add(heap.RemoveMin());
10     return list;
11 }
12
13 var ints = HeapSort(new[] {9, 8, 7, 6, 5, 4, 3, 2, 1});
14 // Output atteso: 1 2 3 4 5 6 7 8 9
15 foreach (var i in ints)
16     Console.Write(i + " ");
```

3.2.2 Esempi d'uso: heap sort e Dijkstra

Esponiamo ora un paio di esempi; il primo, contenuto nel listato 3.6, descrive come poter implementare una versione *naïve*[7] dell'algoritmo di heap sort; notiamo come sia stata scelta la versione ad un elemento (cioè, la versione dove chiave e priorità coincidono) e come la creazione e l'uso dello heap siano intuitivi.

Invece, nel listato 3.7, si può osservare un esempio d'uso leggermente più complicato, nel quale, oltre alle operazioni di aggiunta e di rimozione, si usa anche il cambiamento di priorità (riga 30). Il cambiamento di priorità, come accennato in precedenza, richiede che venga passato un *handle*.

3.2.3 Confronto con le code a priorità di Java

La libreria standard della piattaforma Java contiene un'implementazione della struttura coda a priorità basata sullo heap binario a rappresentazione implicita, e richiede che valore e priorità coincidano nello stesso tipo. La nostra libreria, come spiegato precedentemente, non pone questo vincolo.

Le operazioni esposte dall'implementazione in Java permettono di aggiungere e rimuovere elementi, oltre che verificarne la presenza all'interno della struttura. Hippie offre anche la possibilità di modificare la priorità di elementi aggiunti allo heap e di unire, in modo più o meno efficiente, due heap esistenti.

Riassumendo, Hippie offre:

- Diversi tipi di heap, mentre in Java vi è una sola implementazione.

[7] L'algoritmo dell'esempio viene definito come *naïve* in quanto l'originale algoritmo di heap sort non usa *direttamente* uno heap per effettuare l'ordinamento, ma sfrutta l'idea sottostante lo heap basato su array per ordinare una certa collezione [90].

Listato 3.7 Implementazione su Hippie dell'algortimo di Dijkstra.

```
1  struct Edge {
2      public int Length;
3      public int Target;
4  }
5
6  int[] Traverse(IRawHeap<int, int> heap,
7                 IList<IEnumerable<Edge>> edges,
8                 int start, int nodeCount)
9  {
10     var distances = new int[nodeCount];
11     var visited = new bool[nodeCount];
12     var nodes = new IHeapHandle<int, int>[nodeCount];
13     for (var i = 0; i < nodeCount; ++i) {
14         nodes[i] = heap.Add(i, int.MaxValue);
15         distances[i] = int.MaxValue;
16     }
17     heap[nodes[start]] = 0;
18
19     while (heap.Count != 0) {
20         var u = heap.RemoveMin();
21         if (u.Priority == int.MaxValue) break;
22         var uId = u.Value;
23         distances[uId] = u.Priority;
24         visited[uId] = true;
25         foreach (var e in edges[uId]) {
26             if (visited[e.Target]) continue;
27             var tmpDist = u.Priority + e.Length;
28             var v = nodes[e.Target];
29             if (tmpDist < v.Priority)
30                 heap[v] = tmpDist;
31         }
32     }
33     return distances;
34 }
```

- Un maggior numero di operazioni, oltre a un maggior numero di interfacce utili per modellare i diversi casi d'uso.

3.2.4 Confronto con C5

Anche se non è molto conosciuta, la libreria C5 [91] implementa una quantità notevole di strutture dati, migliorando quelle già presenti nella libreria standard e aggiungendone di nuove, tra le quali le code a priorità. Come nel caso di Java, la struttura scelta per realizzare lo heap è una sola ed è la struttura *interval heap* [92], la quale permette di accedere all'elemento minimo e massimo in tempo costante.

Detto ciò, e vista la professionalità della libreria C5, è facile intuire che le funzionalità offerte da tale libreria siano ben superiori a quelle di Hippie. Infatti, C5 consente di registrare dei gestori per gli eventi che sono attivati dalle varie operazioni (elemento aggiunto, elemento rimosso, ecc ecc), concetto che in Hippie non è assolutamente presente.

In ogni caso, si è scelto di realizzare Hippie, nonostante C5 fosse già una valida scelta, sia per questioni di studio, come affrontare il design e la pubblicazione di una libreria fortemente generica, sia per avere una maggiore facilità d'uso. Hippie, infatti, è stata pensata in modo da fornire un ottimo compromesso tra funzionalità offerte, semplicità d'uso e prestazioni.

3.3 Libreria Troschuetz.Random

Quando si scrive una simulazione è spesso necessario studiarne il comportamento al variare di certi parametri. Ad esempio, supponendo di voler studiare i tempi di attesa dei clienti in un negozio, si potrebbe valutare come tale valore cambi al variare della frequenza di arrivo dei clienti. Tuttavia, sarebbe irrealistico pensare che i clienti arrivino ogni X minuti, mentre ci si avvicina maggiormente alla realtà se si pensa che i clienti arrivino *in media* ogni X minuti. Già da questo piccolo esempio, si vede la necessità di avere a disposizione un generatore di numeri casuali, in grado di emetterli anche secondo particolari distribuzioni di probabilità: ad esempio, nella simulazione appena descritta, sarebbe opportuno modellare la frequenza degli arrivi con una distribuzione esponenziale avente X come media [93].

La libreria standard di .NET, analogamente al caso degli heap, è priva di generatori di numeri casuali secondo distribuzioni date. L'unico generatore all'interno di tale libreria, la classe `System.Random` [94], può solo generare numeri con distribuzione uniforme. Al contrario, Python offre un generatore ben più potente, presente nel modulo `random` [95], con il quale è possibile ottenere, in modo molto elegante, numeri distribuiti secondo diverse distribuzioni di probabilità. Il listato 3.8 mostra come si possa utilizzare tale modulo per ricavare numeri secondo le distribuzioni esponenziale e normale, mentre il listato 3.9 contiene un possibile output di tale script.

Listato 3.8 Esempio d'uso del modulo random di Python.

```
1 import random
2
3 # Generiamo tre numeri casuali aventi una
4 # distribuzione esponenziale, con 5 come media.
5 print("Exponential:")
6 for i in range(3):
7     print(random.expovariate(1.0/5.0))
8
9 # Generiamo tre numeri casuali aventi una
10 # distribuzione normale, con 7 come media
11 # e 0.5 come varianza.
12 print("Normal:")
13 for i in range(3):
14     print(random.normalvariate(7.0, 0.5))
```

Listato 3.9 Un possibile output dello script in 3.8.

```
Exponential:
8.16173211219
6.96833311322
1.44971992699
Normal:
6.38197094054
6.67449972009
6.49374762075
```

Pertanto, vista la necessità di diversi generatori di numeri casuali e la loro assenza dalla libreria standard, abbiamo cercato su internet delle possibili alternative. In particolare, abbiamo rinvenuto un vecchio articolo pubblicato nel 2006 su *Code Project* [96], nel quale veniva esposta una libreria, *Troschuetz.Random* [97], la quale sembrava soddisfare le nostre necessità. In particolare, in essa erano implementate una quantità considerevole di distribuzioni di probabilità[8] e quattro diversi generatori di numeri casuali, alcuni dedicati alla velocità di generazione e altri dedicati alla precisione.

Considerato che tale libreria era open source, ne abbiamo preso il codice e abbiamo:

- Migliorato la documentazione e ristrutturato il codice sottostante, cercando di introdurre opportune interfacce e modifiche senza danneggiare eccessivamente la retro compatibilità.
- Aggiunto una nuova classe, `TRandom`, la quale espone tutta una serie di metodi in modo tale che essa sia facile da usare, ma allo stesso tempo potente, come il modulo `random` di Python.
- Aggiunto una nuova distribuzione di probabilità, la distribuzione discreta *categorica* [98].
- Scritto una batteria di *unit test*, che ci ha anche permesso di individuare un errore nella distribuzione `Erlang`. Il testing di una libreria per la generazione di numeri casuali ha presentato diverse sfide, motivo per cui alcuni test, tutt'ora, non danno risultati consistenti.
- Preparato i pacchetti per NuGet e li abbiamo pubblicati su tale sito.

Prima di passare alla descrizione della libreria successiva, vedremo ancora alcuni dettagli sulla classe TRandom e spenderemo due parole per confrontare la libreria Troschuetz.Random con la più diffusa *Math.NET Numerics*.

3.3.1 Classe `TRandom`

Spinti dalla necessità di avere a disposizione un qualcosa di analogo al modulo random di Python, abbiamo deciso di creare una classe, `TRandom`, che offrisse lo stesso insieme di funzionalità, pur mantenendo una grande semplicità d'uso. Senza l'introduzione di tale classe, usando soltanto la libreria Troschuetz.Random, l'esempio in 3.8 si sarebbe potuto scrivere come mostrato nel listato 3.10; si osserva facilmente che l'eleganza del modulo Python sarebbe immediatamente persa.

Inoltre, supponendo che in una stessa simulazione si usino diverse distribuzioni di probabilità, magari dello stesso tipo con differenti parametri, allora si può intuire la necessità di una classe che offra l'accesso alle distribuzioni in modo più sintetico. Infatti, seguendo l'approccio usato nell'esempio sopra, dovremmo

[8] Cinque distribuzioni discrete (Bernoulli, Binomiale, Discreta Uniforme, Geometrica, Poisson) e ben *venti* distribuzioni continue (tra cui Normale, Esponenziale, Chi Quadro, Beta).

Listato 3.10 Esempio in 3.8 riscritto usando l'originale Troschuetz.Random.

```
1  // Generiamo tre numeri casuali aventi una
2  // distribuzione esponenziale, con 5 come media.
3  var exponential = new ExponentialDistribution(1.0/5.0);
4  Console.WriteLine("Exponential:");
5  for (var i = 0; i < 3; ++i)
6      Console.WriteLine(exponential.NextDouble());
7
8  // Generiamo tre numeri casuali aventi una
9  // distribuzione normale, con 7 come media
10 // e 0.5 come varianza.
11 var normal = new NormalDistribution(7.0, 0.5);
12 Console.WriteLine("Normal:");
13 for (var i = 0; i < 3; ++i)
14     Console.WriteLine(normal.NextDouble());
```

definire una variabile per ogni differente distribuzione, con eventuali ripetizioni nel caso di differenti parametri; cosa ancora peggiore, quelle variabili potrebbero dover essere usate in più parti della simulazione e il loro numero elevato ne renderebbe la condivisione sicuramente poco pratica.

La classe `TRandom` fa alcuni sacrifici in termini di efficienza spazio-temporale proprio per evitare situazioni simili, esponendo un'interfaccia semplice da usare e molto simile al modulo `random` di Python.

Ad esempio, usando la classe da noi definita, lo script sopra esposto può essere riscritto come nel listato 3.11 o come nel listato 3.12, nel quale si adopera un'interfaccia *fluent* [99].

Inoltre, abbiamo introdotto un metodo per scegliere un elemento casuale da una collezione; il metodo si chiama `Choice` e può essere applicato su collezioni che implementano l'interfaccia `IList<T>` [100], proprio come viene fatto nell'esempio in 3.13.

Quindi, riteniamo che la classe `TRandom`, per quanto concettualmente semplice, possa rivelarsi un elemento chiave per facilitare la stesura di simulazioni. Infatti, l'accesso rapido, ma soprattutto *centralizzato*, a un considerevole numero di distribuzioni è uno strumento comodo e potente per chi si accinge a scrivere simulazioni utilizzando la libreria Dessert.

3.3.2 Confronto con Math.NET Numerics

La libreria Math.NET Numerics [101] nasce con lo scopo di colmare un grosso vuoto all'interno della piattaforma .NET, fornendo una serie di strumenti utili per la computazione ai fini di analisi numerica e scientifica. Abbiamo già evidenziato il fatto che, nella libreria standard, mancano dei validi generatori

Listato 3.11 Esempio in 3.8 riscritto usando la classe `TRandom`.

```
var rand = new TRandom();

// Generiamo tre numeri casuali aventi una
// distribuzione esponenziale, con 5 come media.
Console.WriteLine("Exponential:");
for (var i = 0; i < 3; ++i)
    Console.WriteLine(rand.Exponential(1.0/5.0));

// Generiamo tre numeri casuali aventi una
// distribuzione normale, con 7 come media
// e 0.5 come varianza.
Console.WriteLine("Normal:");
for (var i = 0; i < 3; ++i)
    Console.WriteLine(rand.Normal(7.0, 0.5));
```

Listato 3.12 Esempio in 3.8 riscritto usando l'interfaccia *fluent* di `TRandom`.

```
var rand = new TRandom();

// Generiamo tre numeri casuali aventi una
// distribuzione esponenziale, con 5 come media.
Console.WriteLine("Exponential:");
foreach (var e in rand.ExponentialSamples(1.0/5.0).Take(3))
    Console.WriteLine(e);

// Generiamo tre numeri casuali aventi una
// distribuzione normale, con 7 come media
// e 0.5 come varianza.
Console.WriteLine("Normal:");
foreach (var n in rand.NormalSamples(7.0, 0.5).Take(3))
    Console.WriteLine(n);
```

Listato 3.13 Uso del metodo `Choice` esposto da `TRandom`.

```
var names = new[] {"Pino", "Gino", "Dino"};
var rand = new TRandom();

// Estraiamo un nome a caso da "names"
// e lo stampiamo sulla console.
Console.WriteLine("Nome scelto: {0}", rand.Choice(names));
```

di numeri casuali; oltre a quelli, manca ad esempio il supporto per il calcolo vettoriale e matriciale e le trasformate di Fourier [102].

La parte relativa ai numeri casuali è stata ottenuta modificando il codice della libreria Troschuetz.Random, come nel nostro caso. Tuttavia, mentre noi abbiamo cercato di mantenere il più possibile la compatibilità con la versione originale, compresa l'attribuzione della licenza, il progetto Math.NET Numerics ha apportato modifiche più estensive, come è possibile osservare analizzando i sorgenti del progetto [103].

Le ragioni per cui è stato scelto di ristrutturare Troschuetz.Random, piuttosto che integrare Math.NET Numerics, sono principalmente due; la prima è che la libreria Troschuetz.Random è unicamente dedicata alla generazioni di numeri casuali: ciò la rende sia più facile da comprendere per l'utente, sia più facile da mantenere. La seconda ragione è che abbiamo ritenuto un'esperienza interessante poter mettere mano a una libreria seria e funzionante, con lo scopo di renderla più moderna pur mantenendo la retro compatibilità. Inoltre, il poter mettere mano ai sorgenti ci ha concesso la possibilità di integrare la classe TRandom direttamente all'interno della nostra libreria, cosa che non sarebbe stata possibile con Math.NET Numerics.

3.4 Libreria Thrower

Quando si scrivono librerie destinate all'uso pubblico, risulta necessario far sì che siano ben documentate e impediscano all'utente di usarle in modo "scorretto". In ambito .NET il secondo obiettivo si ottiene lanciando opportune e documentate eccezioni nel momento in cui l'utente della libreria invoca un comando in modo errato.

I seguenti esempi mostrano come, dalle ragioni sopra esposte, siamo arrivati alla scrittura di una libreria, Thrower, dedicata unicamente alla facilitazione della scrittura dei controlli di integrità e usata con successo da tutte le altre librerie da noi create.

Nel listato 3.14 definiamo una semplice interfaccia che modella una banca e, con essa, vedremo vari modi di implementare i controlli.

3.4.1 Implementazione classica

Chiunque dovesse implementare rapidamente una classe conforme all'interfaccia `IBank`, mostrata nel listato 3.14, senza ombra di dubbio realizzerebbe i controlli come una serie di `if`, ottenendo del codice simile a quello riportato nel listato 3.15.

Premesso che questo approccio non ha assolutamente nulla di scorretto, vorremmo tuttavia vedere se sia possibile migliorarlo o meno. In particolare, notiamo due cose: a livello estetico, il codice è piuttosto tedioso da leggere e da scrivere; nel caso di molti vincoli, la sezione dedicata agli `if` potrebbe raggiungere lunghezze notevoli. A livello di rapidità di esecuzione, invece, osserviamo che non vi è modo di sopprimere quegli `if`, a meno di commentarli.

Listato 3.14 Semplice interfaccia per modellare una banca.

```
1  interface IBank {
2      /// <summary>
3      ///   La cifra depositata nella banca.
4      /// </summary>
5      double Amount { get; }
6
7      /// <summary>
8      ///   Vero se la banca è aperta, falso altrimenti.
9      /// </summary>
10     bool IsOpen { get; }
11
12     /// <summary>
13     ///   Deposita la quantità data nella banca.
14     /// </summary>
15     /// <param name="amount">
16     ///   Una quantità positiva di denaro.
17     /// </param>
18     /// <exception cref="ArgumentOutOfRangeException">
19     ///   La quantità è zero o negativa.
20     /// </exception>
21     /// <exception cref="InvalidOperationException">
22     ///   La banca è chiusa.
23     /// </exception>
24     void Deposit(double amount);
25 }
```

Listato 3.15 Implementazione immediata delle specifiche delle banca.

```
// Per rendere lo script più sintetico...
using IOP = System.InvalidOperationException;
using AOR = System.ArgumentOutOfRangeException;

class MyBank : IBank {
    public double Amount { get; private set; }
    public bool IsOpen { get; private set; }

    public void Deposit(double amount) {
        if (!IsOpen)
            throw new IOP("La banca è chiusa");
        if (amount <= 0)
            throw new AOR("Quantità non positiva");
        Amount += amount;
    }
}
```

Il lettore si starà chiedendo perché, dopo tutta la fatica fatta per scrivere i controlli, li vorremmo togliere. La ragione è molto semplice: se sappiamo per esperienza, o tramite opportuno testing, che non violeremo mai i vincoli di una libreria, perché gettare via cicli di clock per controllarli? Talvolta, i controlli possono essere computazionalmente pesanti (come controllare la presenza di un elemento in una collezione) e, nel caso in cui il codice della libreria venga invocato con altissima frequenza, proprio quei controlli potrebbero ridurre le prestazioni.

Per la maggior parte delle applicazioni, i cicli di clock guadagnati non sarebbero così rilevanti, ma nel caso del simulatore, il nostro progetto, lo sono. Il nostro motore di simulazione si appoggia a Hippie e Slinky, usando avidamente le strutture dati da loro esposte; in particolare, quasi ogni evento implica un'interazione con le code a priorità sottostanti. Nel caso di simulazioni su larga scala, vorremmo che il motore girasse il più veloce possibile, senza perdere tempo in controlli inutili.

Così, dopo alcune ricerche, abbiamo rinvenuto la libreria *Code Contracts*, sulla quale andrebbe dedicata una relazione intera ma noi ci limiteremo a farne un breve cenno.

3.4.2 Implementazione con Code Contracts

Code Contracts [104] è un vero e proprio insieme di strumenti, offerto da Microsoft, per agevolare la scrittura di codice *robusto*. Infatti, tali strumenti facilitano la scrittura dei controlli di integrità disaccoppiandoli dal codice dei metodi e of-

Listato 3.16 Interfaccia annotata per Code Contracts.

```
1  [ContractClass(typeof(BankContract))]
2  interface IBank {
3      /// <summary>
4      ///   ...
5      /// </summary>
6      [Pure]
7      double Amount { get; }
8
9      /// <summary>
10     ///   ...
11     /// </summary>
12     [Pure]
13     bool IsOpen { get; }
14
15     /// <summary>
16     ///   ...
17     /// </summary>
18     void Deposit(double amount);
19 }
```

frono degli analizzatori di correttezza che funzionano sia a tempo di esecuzione, sia a tempo di compilazione.

Per usare al meglio Code Contracts, occorre inizialmente arricchire l'interfaccia definita per la banca, annotandola con opportuni attributi definiti all'interno di Code Contracts (listato 3.16).

Dopodiché, si definisce una classe dedicata unicamente alla gestione dei controlli di integrità. Tale classe dovrà essere astratta, dovrà implementare l'interfaccia della banca e, all'interno dei metodi, potrà soltanto usare delle chiamate a Code Contracts; ciò viene realizzato nel listato 3.17. Fatto ciò, l'implementazione della nostra banca diventa semplicemente ciò che è riportato nell'esempio 3.18.

Gli specifici controlli verranno "iniettati" direttamente da Code Contracts durante la compilazione; in particolare, è possibile controllare quali parti del contratto verranno iniettate: ad esempio, nella fase di *debug* sarà opportuno aggiungere tutto, mentre nella fase di rilascio si lasceranno soltanto i controlli inerenti le eccezioni documentate. Oppure, se si vuole, si possono omettere del tutto i controlli.

Quindi, Code Contracts rispecchia effettivamente i nostri requisiti, dato che ci da la possibilità di avere dei controlli disabilitabili e ci consente di esprimerli in maniera concisa, elegante e pratica. Nonostante ciò, abbiamo deciso di scrivere Thrower, poiché vorremmo che il nostro progetto fosse compilabile anche tramite la piattaforma open source *Mono* [105] che offre soltanto un supporto parziale

Listato 3.17 Contratto per l'interfaccia della banca.

```
1  // Per rendere lo script più sintetico...
2  using IOP = System.InvalidOperationException;
3  using AOR = System.ArgumentOutOfRangeException;
4
5  [ContractClassFor(typeof(IBank))]
6  abstract class BankContract {
7      double Amount {
8          get {
9              Contract.Ensures(Contract.Result<double>() >= 0);
10             return default(double);
11         }
12     }
13
14     // Non si definisce un particolare contratto
15     abstract bool IsOpen { get; }
16
17     void Deposit(double amount) {
18         Contract.Requires<IOP>(IsOpen,
19                                "La banca è chiusa");
20         Contract.Requires<AOR>(amount > 0,
21                                "Quantità non positiva");
22         Contract.Ensures(Amount ==
23                          Contract.OldValue(Amount) + amount);
24     }
25 }
```

Listato 3.18 Implementazione della banca usando i contratti.

```
1  class MyBank : IBank {
2      public double Amount { get; private set; }
3      public bool IsOpen { get; private set; }
4
5      public void Deposit(double amount) {
6          Amount += amount;
7      }
8  }
```

Listato 3.19 Implementazione della banca usando Thrower.

```
1 // Per rendere lo script più sintetico...
2 using IOP = System.InvalidOperationException;
3 using AOR = System.ArgumentOutOfRangeException;
4
5 class MyBank : IBank {
6     public double Amount { get; private set; }
7     public bool IsOpen { get; private set; }
8
9     public void Deposit(double amount) {
10        Raise<IOP>.IfNot(IsOpen, "La banca è chiusa");
11        Raise<AOR>.If(amount <= 0, "Quantità non positiva");
12        Amount += amount;
13    }
14 }
```

ai contratti [106]. Inoltre, al momento, gli IDE open source, MonoDevelop e SharpDevelop, non ne offrono proprio [107, 108].

3.4.3 Implementazione con Thrower

Thrower è la nostra risposta al problema evidenziato all'inizio: avere dei controlli rimovibili, facili da scrivere e utilizzabili anche su Mono.

La nostra libreria, molto piccola e semplice, offre tutta una serie di metodi con i quali poter descrivere in modo *fluent* i controlli di integrità, come nell'esempio seguente presentato nel listato 3.19.

La classe statica `Raise<TEx>` contiene i metodi con cui poter effettuare i controlli e si occupa lei stessa, tramite *reflection*, di creare e lanciare le eccezioni quando richiesto. I metodi di quella classe sono marcati con l'attributo `ConditionalAttribute` [109] e sono attivi solo quando viene definita la costante di compilazione `USETHROWER`. In caso contrario, i metodi semplicemente "scompaiono", in modo analogo a ciò che succede con le chiamate a `Debug.Assert` [110].

3.5 Libreria Slinky

Ci soffermeremo molto brevemente sulla libreria Slinky, in quanto è sorta solamente per risolvere un problema triviale presente nella libreria standard di .NET. L'unica implementazione di liste "linkate" offerta, `LinkedList<T>` [111], espone la struttura usata per i nodi, con l'apparente scopo di velocizzare operazioni come la rimozione e il contenimento. Tuttavia, esponendo direttamente i nodi, alcuni metodi, come l'unione di due liste, non sono esposti perché non avrebbero le complessità che potrebbero avere ($O(n)$ invece $O(1)$).

La causa di tutto ciò è che, per mantenere al meglio l'integrità, ciascun nodo contiene un riferimento alla lista a cui appartiene. Questo fatto, oltre a rappresentare un impiccio dal punto di vista del consumo di memoria, lo rappresenterebbe anche nell'unione. Mentre per una comune lista "linkata" sarebbe possibile realizzare l'operazione di unione in tempo costante lavorando sui puntatori al primo e all'ultimo elemento, l'implementazione .NET deve:

1. Prendere ciascun nodo della lista che si sta unendo e correggerne il riferimento;
2. Aggiustare i puntatori al primo e all'ultimo elemento della lista principale.

Quindi, Slinky si occupa di questo problema operando su più fronti:

- Vengono definiti molti tipi di liste "linkate", dalle più semplici e leggere, dove si memorizzano solo il puntatore al primo elemento e solo il puntatore al successivo in ciascun nodo, sino alle più potenti e pesanti, con puntatori a primo e ultimo elemento, successore e predecessore nei nodi, e anche una mappa *(valore, nodo)* per velocizzare alcune operazioni.
- L'unione, quando possibile, è realizzata in tempo costante.
- Sono anche implementati delle code e degli stack basati sulle liste "linkate", frequentemente utilizzati all'interno del simulatore perché hanno un basso consumo di memoria e, per piccole collezioni, sono più efficienti delle controparti basate su array.

La libreria Slinky è altamente usata da Hippie per poter implementare gli heap basati sulla rappresentazione esplicita ad albero, dove le liste sono usate per mantenere i figli degli alberi, che hanno spesso un numero variabile di elementi.

3.5.1 Esempi d'uso

Per dare un'idea più concreta su Slinky, proponiamo alcuni semplici esempi scritti in F#[9]. Nel listato 3.20 si può osservare un utilizzo della lista linkata di tipo *thin*, la quale, sacrificando l'esposizione di alcune operazioni, è molto efficiente dal punto di vista del consumo di memoria.

Nel listato 3.21 si usano le funzionalità dello stack basato su liste linkate che Slinky offre.

Gli esempi sono molto semplici, indubbiamente, ma ciò è inevitabile vista la semplicità intrinseca della libreria Slinky: come è stato detto, la sua ragion d'essere era quella di porre rimedio a un problema molto ben circoscritto dell'implementazione delle liste "linkate" in .NET.

[9] Altri esempi scritti in quel linguaggio verranno proposti nel corso di tutta la relazione. Lo scopo è duplice: primo, far conoscere un linguaggio che è sia funzionale, sia "pratico", nel senso che alcuni aspetti comodi dei linguaggi imperativi (cicli, classi, mutabilità) vengono preservati; secondo, dimostrare come la piattaforma .NET consenta di scrivere librerie che potranno essere facilmente usate da più linguaggi, anche se seguono paradigmi differenti.

Listato 3.20 Esempio d'uso delle liste linkate leggere esposte da Slinky.

```
open Slinky

// Le liste di tipo "thin" hanno il minore consumo
// di memoria, ma offrono un numero esiguo di
// operazioni applicabili su di esse.
let tl = ListFactory.NewThinLinkedList<string>()

tl.Add("terzo")
tl.Add("secondo")
tl.Add("primo")
// Stampa attesa: primo secondo terzo
for i in tl do printf "%s " i
printfn ""

tl.RemoveFirst()
// Stampa attesa: secondo terzo
for i in tl do printf "%s " i
printfn ""

tl.AddFirst("primo")
tl.Remove("secondo") |> ignore
// Stampa attesa: primo terzo
for i in tl do printf "%s " i
printfn ""
```

Listato 3.21 Esempio d'uso dello stack offerto da Slinky.

```
1  open Slinky
2
3  // Si ottiene uno stack implementato
4  // su una lista linkata di tipo thin.
5  let ls = ListFactory.NewLinkedStack<string>()
6
7  ls.Push("terzo")
8  ls.Push("secondo")
9  ls.Push("primo")
10 // Stampa attesa: primo secondo terzo
11 for i in ls do printf "%s " i
12 printfn ""
13
14 ls.Pop() |> ignore
15 ls.Pop() |> ignore
16 // Stampa attesa: terzo
17 printfn "%s" ls.Top
```

Capitolo 4

Libreria Dessert

Finalmente, dopo le opportune premesse e le varie descrizioni introduttive, possiamo passare al racconto per esteso del nostro progetto, Dessert. I sorgenti del progetto, il cui sviluppo è stato reso pubblico da lungo tempo, sono consultabili in [112]; inoltre, abbiamo già caricato alcune versioni di Dessert su NuGet [113], per facilitare il lettore nella sperimentazione di quanto racconteremo e nell'estensione degli esempi proposti.

Nel listato 4.1 presentiamo un semplice esempio d'uso della libreria Dessert: la traduzione dell'esempio, per SimPy, mostrato nel listato 2.1, cosicché sia possibile fare un rapido confronto tra i due *linguaggi di simulazione* e si possa capire che, esclusi i "fronzoli" dovuti al linguaggio di programmazione scelto, i concetti introdotti da SimPy sono stati esattamente ripresi dal nostro lavoro.

Questo capitolo descrive un gran numero di dettagli che, per forza di cose, abbiamo lasciato in sospeso nei capitoli precedenti. Precisamente, in sezione 4.1 parleremo dei principi di design e in sezione 4.2 delle scelte tecniche che abbiamo fatto, e di come le diverse librerie presentate nel capitolo 3 siano state integrate all'interno del progetto.

Poi, rispettivamente nelle sezioni 4.3 e 4.4, vedremo come la nostra implementazione si rapporti, in termini di interfacce esposte e di modalità d'uso, con l'originale libreria SimPy. Per i confronti a livello di prestazioni, invece, rimandiamo il lettore al capitolo 5.

Infine, in sezione 4.5, riscriveremo l'esempio della banca, che abbiamo già implementato su SimPy nella sezione 2.6. Al fine di non appesantire eccessivamente il contenuto di questo capitolo, le traduzioni degli altri esempi presentati nel capitolo 2, anch'essi tradotti per poter essere eseguiti su Dessert, saranno riportate in appendice B.

4.1 Principi di design

Prima di iniziare lo sviluppo abbiamo stilato una semplice *lista* di principi di design che avremmo voluto seguire nel corso dell'implementazione. Le ragioni

Listato 4.1 Semplice esempio d'uso di Dessert.

```
1  open Dessert
2
3  let rec car(env: IEnvironment) = seq<IEvent> {
4      printfn "Start parking at %g" env.Now
5      let parkingDuration = 5.0
6      yield upcast env.Timeout(parkingDuration)
7
8      printfn "Start driving at %g" env.Now
9      let tripDuration = 2.0
10     yield upcast env.Timeout(tripDuration)
11
12     // Esegue nuovamente lo stesso processo.
13     // L'ottimizzazione della ricorsione di coda
14     // rende il seguente comando simile a un "while true".
15     yield! car(env)
16 }
17
18 let env = Sim.NewEnvironment()
19 env.Start(car(env)) |> ignore
20 env.Run(until = 15.0)
```

dietro a tale scelta sono state la volontà di avere uno sviluppo omogeneo della libreria, dove le interfacce seguissero il medesimo stile, e la volontà di avere non un mero clone di SimPy, ma una libreria che potesse avere una propria identità e proprie caratteristiche peculiari.

4.1.1 Rapporto con SimPy

Premesso che non siamo esperti di simulazione, men che meno di progettazione di simulatori, abbiamo deciso di seguire il più possibile il design di SimPy; pertanto, Dessert avrebbe dovuto riproporre gli stessi concetti, possibilmente con nomi analoghi e con analoghe modalità d'uso. Chiaramente, ciò doveva essere mediato da altri principi, come la buona *cittadinanza* in ambiente .NET, che descriveremo a breve, e il tutto doveva essere adattato alle limitazioni tecniche della piattaforma. Ad esempio, avremmo dovuto riprogettare tutti quegli aspetti di SimPy che si basavano sul fatto che `yield`, in Python, sia un'espressione e non soltanto uno *statement*, come lo è in C#.

Tuttavia, ribadiamo il fatto che, nel limite del possibile, Dessert ha cercato di essere simile a SimPy: pertanto, chi ha praticità con lo sviluppo di simulazioni con SimPy non troverà particolari barriere nello spostamento su Dessert, e viceversa. Infatti, questo era uno degli scopi secondari del progetto, cioè, far sì che passare da una libreria all'altra non comportasse ulteriore sforzo di ap-

prendimento da parte dello sviluppatore, il quale doveva poter essere in grado di trasferire facilmente la sua conoscenza del paradigma a processi esposto da SimPy.

4.1.2 Integrazione in ambiente .NET

Uno degli aspetti più famosi della piattaforma .NET è la possibilità di utilizzare una libreria, scritta in un linguaggio per tale piattaforma, da un qualunque altro linguaggio. Per facilitare la portabilità esiste la cosiddetta *CLS compliancy* [114], la quale stabilisce varie convenzioni, tra cui diverse linee guida da seguire nella scelta dei nomi per i metodi e le classi. Per far capire l'importanza pratica di ciò, occorre ricordare che Visual Basic .NET non è *case sensitive* [115]: perciò, nomi validi in C#, come `If` e `Throw`, diventano parole chiave nell'altro linguaggio.

Pertanto, abbiamo cercato di far sì che Dessert seguisse tali convenzioni, in modo tale che fosse possibile utilizzarla da un gran numero di linguaggi. Sperabilmente, questa scelta dovrebbe ampliare il possibile bacino d'utenza della libreria, portando la simulazione a eventi discreti verso linguaggi che, per quanto ci è noto alla scrittura di questa relazione, non dispongono di altre librerie dedicate a tale scopo.

Nel repository del progetto, pubblicamente visitabile in [112], abbiamo scritto diversi esempi usando un buon numero di linguaggi per .NET, al fine di dimostrare che la nostra libreria è effettivamente usabile da svariati linguaggi e che, in generale, il suo uso continua a essere intuitivo in ciascuno di essi. In particolare, gli esempi, per un certo linguaggio *LANG*, sono memorizzati all'interno della cartella *Dessert.Examples.LANG*.

4.1.3 Introduzione dei tipi

La libreria SimPy, essendo stata progetta per, e scritta in, Python, non ha bisogno di introdurre e sfruttare la nozione di tipo statico; al contrario, Dessert ne deve tenere conto, per poter rendere più veloce e robusto l'uso della libreria stessa. Infatti, i tipi statici ci consentono di evitare l'uso dei cast e permettono al compilatore di rilevare tutta una serie di errori che, in Python, possono essere solo scoperti a tempo di esecuzione.

Le interfacce esposte da SimPy, per essere correttamente tradotte su .NET, devono essere annotate con opportuni tipi. Inoltre, a volte è possibile migliorarle o arricchirle in modo che offrano maggiori informazioni sui tipi coinvolti. Per capire meglio cosa intendiamo, si consideri la risorsa Store di SimPy: se la implementassimo esattamente come viene esposta da SimPy, avremmo che, quando si inserisce un oggetto al suo interno, si perderebbe ogni informazione di tipo. Quindi, per esempio, non avremmo la possibilità di filtrare staticamente i tipi inseriti, perché non vi sarebbe alcun vincolo su di essi; il listato 4.2 esprime, tramite alcune righe di codice, quale sia il problema che vorremmo evitare. Pertanto, nella nostra *traduzione* è stato necessario introdurre l'uso dei tipi generici, in modo da colmare le mancanze appena evidenziate.

Listato 4.2 Negli Store di SimPy non si può specificare il tipo degli elementi.

```
1 def consumer(env, stringStore):
2     while True:
3         msg = yield stringStore.get()
4         print("First char of message: %c" % msg[0])
5         # La prima iterazione è OK, stamperà "P".
6         # Tuttavia, si avrà un crash per la seconda,
7         # poiché un intero non è indicizzabile.
8
9 env = Environment()
10 stringStore = Store(env)
11 stringStore.put("PINO") # Una stringa è OK
12 stringStore.put(21) # Ops, abbiamo inserito un intero!
13 env.start(consumer(env, stringStore))
14 env.run()
```

4.1.4 *Armando*, il layer di traduzione verso SimPy

Come estensione al nostro progetto, abbiamo deciso di creare, tramite l'uso di IronPython [77], una sorta di *layer* col quale fosse possibile eseguire simulazioni per SimPy sul nostro motore, Dessert. L'idea era quella di provare a vedere se, eseguendo il tutto su Dessert, si riuscisse in qualche modo a migliorare le prestazioni delle simulazioni scritte per SimPy, senza modificarne una singola riga di codice. Da tale idea è nato il progetto Armando, con il quale avremmo esplorato la fattibilità delle nostre intenzioni.

I risultati del nostro esperimento saranno esposti nel capitolo 5, ma qui ci vorremmo soffermare sulle scelte di progettazione. Senza dilungarci eccessivamente in dettagli relativi a IronPython, ricordiamo che è sia un'implementazione di Python, sia un motore di *scripting*, che consente di utilizzare librerie .NET, con le relative classi, direttamente dagli script Python.

Di fatto, gli oggetti .NET vengono *passati* dal motore all'interno dell'ambiente Python, in modo tale da poter essere usati proprio come se fossero oggetti nativi; IronPython fa abbondante uso della *reflection* per analizzare gli oggetti e scoprire, a tempo di esecuzione, se essi abbiano i campi e gli attributi richiesti dallo script.

Pertanto, per costruire Armando, sarebbe stato sufficiente passare le istanze degli oggetti di Dessert attraverso un opportuno script Python, il quale avrebbe *finto* di essere la libreria SimPy e avrebbe fornito gli oggetti necessari per il buon funzionamento. Tuttavia, per quanto la situazione sembri semplice e ben delineata, ci sono alcuni problemi poco visibili, che hanno rappresentato un ostacolo nella progettazione del layer.

Infatti, come detto prima, abbiamo cercato di far sì che Dessert fosse ben integrata nella piattaforma .NET e che, al tempo stesso, mantenesse il più possibile la nomenclatura introdotta da SimPy. Per realizzare ciò, alcuni nomi

di metodo, come `timeout`, sono rimasti identici, eccetto la maiuscola iniziale (`Timeout`), ma questo ci ha impedito una mappatura diretta dei nostri oggetti verso quelli *attesi* dalle simulazioni per SimPy, le quali si aspettavano nomi minuscoli. Quindi, abbiamo introdotto degli opportuni *overload*, in modo che fossero presenti entrambe le versioni del metodo: tuttavia, per quanto la soluzione fosse funzionante, il risultato ottenuto non era nemmeno lontanamente ottimale. La presenza di diversi overload danneggiava sia la facilità d'uso, in quanto l'utente si sarebbe chiesto quale fosse la differenza tra metodi aventi un nome praticamente identico, sia la possibilità di usare Dessert da VB.NET, dato che in tale linguaggio, non essendo case sensitive, il compilatore non avrebbe saputo quale metodo invocare.

Alla fine, abbiamo raffinato l'idea di fornire diversi overload utilizzando diverse interfacce, in modo tale che gli utenti di Dessert ne ricevessero una, adatta a .NET, e che Armando ne ricevesse un'altra, dove i nomi erano opportunamente adattati alla libreria SimPy.

4.2 Dettagli tecnici

Oltre ai vincoli di progettazione che ci siamo posti autonomamente, descritti in sezione 4.1, abbiamo ulteriormente lavorato affinché la nostra libreria potesse essere utilizzata *realmente* e non restasse una sorta di esperimento. Pertanto, abbiamo adottato diverse scelte tecniche per garantire una maggiore efficienza e ampliare i casi d'uso di Dessert. Ad esempio, abbiamo studiato le performance dovute all'utilizzo degli heap da parte del motore, traendo conclusioni che non erano immaginabili all'inizio dello sviluppo del progetto.

Pertanto, questa sezione tratterà di argomenti prettamente tecnici e focalizzati, più che sulla simulazione, su come ottenere i massimi vantaggi dalla piattaforma .NET e su come sfruttare le sue caratteristiche per ampliare il raggio d'azione di Dessert.

4.2.1 Utilizzo di Hippie

Nel listato 2.3 avevamo presentato una versione decisamente succinta del cuore di un generico simulatore a eventi discreti, dalla quale partiremo per descrivere come gli heap forniti da Hippie siano utilizzati all'interno del motore di Dessert.

Nel nostro esempio, avevamo uno heap, usato come coda a priorità, nel quale inserivamo tutti i processi e i rispettivi tempi di attivazione (implicitamente memorizzati nei processi stessi); fatto ciò, l'ordine parziale indotto dallo heap ci consentiva di sapere quale processo prendere in considerazione in ciascuna iterazione, poiché il processo da eseguire era quello avente il minore tempo di attivazione.

Comprensibilmente, dover implementare un motore complesso come quello di SimPy complica inevitabilmente lo scenario, dato che lì non solo i processi vengono schedulati, ma tutti gli eventi vengono inseriti nella coda a priorità;

tale fatto è dovuto alla scelta di rimuovere la distinzione tra processo ed evento, al fine di uniformare e potenziare l'interfaccia della libreria stessa.

Quindi, dovendo implementare a nostra volta il modello stabilito da SimPy, ci siamo trovati di fronte alla seguente situazione:

- La coda a priorità avrebbe dovuto contenere un gran numero di elementi, divisi tra processi ed eventi; in particolare, la cardinalità dei processi sarebbe stata, grossomodo, equivalente a quella degli eventi, visto che un processo attende in media un evento solo e che, sempre in generale, un evento è atteso da un solo processo.
- Lo heap avrebbe dovuto essere stabile, al fine di garantire un certo determinismo nell'esecuzione delle simulazioni.
- La velocità di inserimento e rimozione, oltre a un basso *overhead* in memoria, sarebbero stati elementi decisivi nel raggiungimento dei nostri obiettivi.

La nostra risposta ai tre requisiti non è esattamente *ovvia*, e per arrivarci abbiamo avuto bisogno di opportune misurazioni, parte della quali saranno presentate nel capitolo 5. In ogni caso, abbiamo per optato per *due* heap binari, che condividono un contatore a 64 bit usato per aggiungere la *stabilità* alle operazioni di aggiunta e rimozione.

Abbiamo scelto di avere due heap, uno per i processi e uno per gli eventi, al fine di ridurre l'altezza degli alberi binari a essi collegati; ciò avrebbe giovato alla rapidità delle operazioni, in quanto esse dipendono unicamente dall'altezza dell'albero binario. Inoltre, array di dimensioni minori avrebbero potuto garantire una maggiore località in memoria e quindi alcuni incrementi nella prestazioni.

Tra i vari heap messi a disposizione da Hippie, abbiamo scelto quelli stabili e di tipo *thin*, che possono garantire migliori tempi di esecuzione e un basso overhead in memoria, poiché offrono un numero esiguo, ma sufficiente, di operazioni.

4.2.2 Uso delle altre librerie

Non ci soffermeremo molto sulla descrizione di dove e come siano utilizzate le altre librerie da noi prodotte, ma ne daremo alcuni cenni al fine di giustificare la loro scrittura.

Thrower viene ampiamente usata per controllare i vincoli di integrità di molti metodi pubblici della libreria Dessert, in modo che sia possibile farli "sparire" in un colpo solo nel caso siano richieste le massime prestazioni. Nel capitolo 5, vedremo tramite opportuni *benchmark* quale sia il reale incremento delle prestazioni che è possibile ottenere.

Relativamente a Troschuetz.Random, Dessert espone una proprietà per ottenere un'istanza della classe `TRandom`, configurata in modo da utilizzare un generatore di tipo *Mersenne twister* [116]; un generatore di quel tipo garantisce un periodo di $2^{19937}-1$, il che lo rende ottimale per simulazioni che "estraggono"

una grande quantità di numeri casuali. Di base, la classe TRandom utilizza un generatore di tipo *Xorshift* [117], che, sebbene disponga di maggiori prestazioni, ha un periodo decisamente più basso (nell'ordine di $2^{128} - 1$).

Infine, Slinky compare in diverse parti del codice, ove servano collezioni con un basso consumo di memoria e contenenti, mediamente, pochi elementi.

4.2.3 Costruzione di Armando

Armando consiste in un ambiente di esecuzione sul quale gli script preparati per SimPy, versione 3, possono essere eseguiti su Dessert, senza modificarne alcuna riga. Ciò è ottenuto tramite un'opportuna combinazione di codice .NET e Python, che si occupa di fornire agli script l'interfaccia necessaria per la loro esecuzione e di mascherare l'uso degli oggetti e dei meccanismi di Dessert.

Il traduttore è un eseguibile .NET, che accetta come parametro il nome del file Python contenente il codice di avvio della simulazione. Di per sé, il traduttore non compie molte operazioni, poiché esse sono necessariamente distribuite nei file che lo accompagnano; in generale, l'eseguibile si occupa soltanto di avviare IronPython e, tramite esso, di leggere e avviare lo script.

Occorre sottolineare il fatto che, allo stato attuale, Armando parte dal presupposto che la simulazione da eseguire sia perfettamente funzionante su SimPy; infatti, non è stata ancora implementata una buona gestione degli errori che consenta lo sviluppo esclusivamente tramite il layer. Di fatto, l'idea dietro al nostro lavoro consiste nel fornire un mezzo per velocizzare l'esecuzione di simulazioni che, su SimPy, potrebbero richiedere troppo tempo. In particolare, IronPython non è vincolato dal *global interpreter lock* [16, 118] e ciò apre la possibilità di eseguire più simulazioni SimPy realmente in parallelo.

Cercheremo di dare una spiegazione rapida, ma al tempo stesso esaustiva, di come Armando sia strutturato. Partiremo con il descrivere il codice di "collegamento", una serie di script Python, per poi passare al traduttore vero e proprio, del codice *ad hoc* scritto su .NET e fortemente basato su IronPython.

4.2.3.1 Interfaccia in Python

Affinché gli script SimPy possano essere eseguiti su Armando senza *alcuna* modifica, risulta necessario fornire loro un'interfaccia che sia identica a quella fornita dall'originale libreria SimPy.

Abbiamo risolto il problema su due fronti, creando da un lato alcuni script Python contenenti i metodi per istanziare classi base come `Environment`, e dall'altro aggiungendo opportuni overload alle nostre classi, in modo che fossero presenti i metodi attesi.

Nel listato 4.3 portiamo un esempio concreto di come l'interfaccia in Python sia strutturata. Per prima cosa, lo script *carica* l'eseguibile contenente il traduttore, `Armando.exe`, poiché esso possiede le classi "ponte" che ci consentiranno di collegarci al motore di Dessert; in particolare, la classe più importante è `CustomMoveHandler`, di cui parleremo a breve, visto che essa sarà utilizzata nel ciclo principale del simulatore.

Listato 4.3 Esempio dell'interfaccia in Python del layer.

```
1  import clr
2  clr.AddReferenceToFile("Armando.exe")
3
4  import sys
5  import Armando
6
7  class Interrupt(Exception):
8      def __init__(self):
9          self.cause = None
10
11 class Environment(object):
12     def __new__(cls):
13         object.__new__(cls)
14         moveHandler = Armando.CustomMoveHandler(Interrupt())
15         return moveHandler.Env
16
17 class Store(object):
18     def __new__(cls, env, capacity = sys.maxint):
19         object.__new__(cls)
20         return env.new_store(capacity)
```

Dopo la parte relativa al caricamento del traduttore, lo script presenta le prime funzioni e classi che saranno usate per far trovare agli script un ambiente compatibile con quello atteso; ad esempio, è possibile notare la definizione della classe `Interrupt`, di cui gli script avranno bisogno per intercettare le interruzioni, e la definizione di alcune classi *stub*, come `Environment` e `Store`. I costruttori di quest'ultime sono i veri metodi *ponte*, perché sono loro a iniettare istanze degli oggetti di Dessert negli script per SimPy.

Quindi, abbiamo dovuto creare una serie di script disposti in modo tale da ricalcare la struttura esposta dalla libreria; ad esempio, parte del listato in 4.3 è memorizzato in `simpy/core.py`, mentre altri file, tra cui possiamo ricordare `simpy/resources/store.py`, contengono gli elementi presenti nella medesima posizione in SimPy.

4.2.3.2 Traduttore basato su IronPython

Per via di come abbiamo deciso di strutturare il traduttore, non è possibile identificarlo né con un singolo file, né con una singola libreria o eseguibile.

Esso è frutto di una *cooperazione* tra elementi sparsi in più file e su più librerie, gestiti da un singolo eseguibile, `Armando.exe`. Esclusa la parte in Python, di cui abbiamo appena discusso, il traduttore consta dei seguenti elementi:

Listato 4.4 Il gestore dell'avanzamento di base.

```
1  SimEvent DefaultMoveHandler(SimProcess process,
2                              SimEvent oldTarget)
3  {
4      if (!process.Steps.MoveNext()) return EndEvent;
5      // Dobbiamo controllare che l'evento restituito
6      // sia stato generato da questa libreria,
7      // poiché dobbiamo essere in grado di
8      // convertirlo al tipo SimEvent.
9      var newTarget = process.Steps.Current as SimEvent;
10     Raise<ArgumentNullException>.IfIsNull(newTarget);
11     return newTarget;
12 }
```

- L'eseguibile, che si occupa dell'avvio di IronPython e della conseguente esecuzione degli script.
- Una serie di interfacce aggiuntive, contenenti opportuni *overload* con lo scopo di fornire l'interfaccia necessaria per gli script.
- Una nuova funzione per gestire l'avanzamento della simulazione, che andrà a sostituire quella di base.
- Ulteriore codice per mappare al meglio i concetti che, per forza di cose, Dessert ha dovuto realizzare in maniera differente rispetto a SimPy.

Dei quattro punti appena delineati, riteniamo che solo uno richieda ulteriori spiegazioni: la funzione per la gestione dell'avanzamento della simulazione, un concetto di cui non abbiamo ancora parlato e che richiede una breve, ma necessaria, descrizione.

Al fine di consentire l'esistenza del traduttore la libreria principale, Dessert, offre la possibilità di ridefinire ciò che abbiamo chiamato *gestore dell'avanzamento*. In pratica, una funzione che riceve in input il processo da eseguire e l'evento che esso stava attendendo, e restituisce l'evento prodotto dall'avanzamento del processo. Per chiarire meglio il concetto, il listato 4.4 contiene l'implementazione di base, inclusa nella libreria Dessert; dal codice, capiamo che, nel caso più semplice, il gestore dovrà semplicemente invocare il metodo `MoveNext` sul generatore corrispondente al processo.

Tuttavia, eseguire gli script Python richiede una maggiore flessibilità, dato che la libreria SimPy usa i generatori in maniera bidirezionale; quindi, non ci possiamo più limitare a spostare in avanti il generatore, ma diventa necessario avere accesso al reale generatore Python sottostante per poter invocare i metodi `send` e `throw`. Per riuscire nel nostro compito, abbiamo dovuto fare uso della reflection e abbiamo dovuto studiare la conformazione degli iteratori restituiti

Listato 4.5 Procedura per ricavare il vero generatore Python.

```
1  Type TW = typeof(IEnumerableOfTWrapper<IEvent>);
2  FieldInfo Generator = TW.GetField("enumerable",
3                                    BindingFlags.Instance |
4                                    BindingFlags.NonPublic);
5
6  IEnumerable GenTransformer(IEnumerable<IEvent> generator)
7  {
8      if (generator is IEnumerableOfTWrapper<IEvent>)
9          return Generator.GetValue(generator) as IEnumerable;
10     // Caso speciale per il processo Until.
11     return generator;
12 }
```

da IronPython, scoprendo che essi contenevano al loro interno un riferimento, privato, al generatore Python.

Pertanto, il tradurre ha comportato l'aggiunta di codice per realizzare le seguenti operazioni:

1. Quando si avvia un processo da Python, si prende l'iteratore restituito e lo si sottopone a una serie di trasformazioni, presentate nel listato 4.5, tramite le quali si ottiene il sottostante generatore Python. Il riferimento a quell'oggetto è memorizzato nel processo stesso.

2. Si ridefinisce il gestore dell'avanzamento, il quale invocherà i metodi `send` e `throw` in modo compatibile con quanto specificato dalla libreria SimPy. La nuova definizione del gestore è mostrata nel listato 4.6.

Il codice corrispondente alla nuova definizione del gestore di avanzamento include anche alcune chiamate alle interfacce di Dessert, di cui non abbiamo ancora parlato, ma dovrebbe comunque essere chiaro il funzionamento della procedura. Brevemente, il generatore Python viene sempre fatto avanzare tramite il metodo `send`, al quale viene passato il valore restituito dall'evento precedente; tuttavia, nel caso ci siano state interruzioni o l'evento precedente sia fallito, si fa avanzare il generatore usando il metodo `throw`, il che produce gli effetti desiderati all'interno degli script per SimPy.

Ricapitolando, il traduttore è un'entità complessa che consta di molte parti, per comprendere le quali occorre anche avere esperienza con la libreria su cui Armando è basato, IronPython. Il traduttore, allo stato attuale, ha forti limitazioni, poiché non è in grado di produrre messaggi di errore significativi e non tutto il codice che potrebbe funzionare su SimPy sarà mai eseguibile sul layer. Per esempio, il cosiddetto *monkey patching* [119], cioè la possibilità in Python di aggiungere attributi e metodi a un oggetto esistente, è una delle principali caratteristiche che non potremo emulare con il nostro traduttore, per via della conformazione più rigida della CLR.

Listato 4.6 Definizione del gestore di avanzamento nel layer.

```
SimEvent MoveHandler(SimProcess process, SimEvent oldTarget)
{
    var gen = process.Generator as PythonGenerator;
    if (gen != null) {
        dynamic next;
        object cause;
        if (Env.ActiveProcess.Interrupted(out cause)) {
            _interrupt.cause = cause;
            next = gen.@throw(_interrupt);
        } else if (oldTarget == null)
            next = gen.send(null);
        else if (oldTarget.Failed)
            next = gen.@throw(oldTarget.Value);
        else
            next = gen.send(oldTarget.Value);
        return (next as SimEvent) ?? Env.EndEvent;
    }
    // Caso speciale per il processo Until.
    var hasNext = process.Steps.MoveNext();
    Debug.Assert(hasNext);
    return process.Steps.Current as SimEvent;
}
```

Nome per SimPy	Nome per Dessert
`Environment`	`IEnvironment`
`Process`	`IProcess`
`Timeout`	`ITimeout<TVal>`
`Event`	`IEvent<TVal>`
`Condition`	`ICondition<T1, ...>`
`Resource`	`IResource`
`PriorityResource`	`IResource`
`PreemptiveResource`	`IPreemptiveResource`
`Container`	`IContainer`
`Store`	`IStore<TItem>`
`FilterStore`	`IFilterStore<TItem>`

Tabella 4.1: Come i nomi usati da SimPy siano stati riportati su Dessert.

4.3 Similarità con SimPy

Uno dei nostri principi era mantenere, nei limiti del possibile, la compatibilità *concettuale* nei confronti di SimPy. Mentre Armando avrebbe provato a mantenere anche una compatibilità sintattica, Dessert, dal canto suo, avrebbe dovuto solo preservare i concetti principali, la loro nomenclatura e, ove possibile, la modalità d'uso.

In generale, siamo riusciti a raggiungere i nostri obiettivi iniziali, come mostreranno vari esempi presentati nel seguito del capitolo e in quelli successivi.

Cominciamo subito a mostrare i punti di contatto in tabella 4.1, dove riportiamo un *breviario* contenente i nomi usati da SimPy e i corrispettivi equivalenti dichiarati da Dessert.

Per quanto riguarda i processi, essi vengono dichiarati come una sequenza di eventi, analogamente a ciò che succede con SimPy; in particolare, Dessert richiede delle sequenze di `IEvent`, un'interfaccia implementata da qualunque elemento possa essere *restituito* da un processo.

Anche le singole interfacce, come auspicabile, riportano metodi e attributi con nomi simili a quelli di SimPy; in tabella 4.2, a titolo esemplificativo, mostriamo come siano stati tradotti gli elementi della classe `Environment`. Alcune parti dell'originale non sono state replicate, vedremo il perché di tale scelta in sezione 4.4.

Nonostante che molto lavoro sia stato fatto perché Dessert fosse, a livello superficiale, decisamente somigliante a SimPy, in sezione 4.4 vedremo che ci sono svariate differenze e aggiunte. Occorre sottolineare il fatto che, nella maggior parte dei casi, le differenze sono dovute a risvolti tecnici e sono state eseguite in modo tale da non portare alla scrittura di più codice del dovuto.

Un piccolo esempio di ciò che intendiamo è dato dalla costruzione dei timeout, che risulta identica a quella di SimPy, nonostante su Dessert l'interfaccia `ITimeout` sia generica. Infatti, supponendo di avere una variabile `env` di tipo `IEnvironment`, costruire un timeout sarà possibile tramite l'istruzione "`var t`

Membri di `Environment`, SimPy	Membri di `IEnvironment`, Dessert
`active_process: Process`	`ActiveProcess: IProcess`
`now: double`	`Now: double`
`start(generator): Process`	`Start(IEnumerable<IEvent>): IProcess`
`process(generator): Process`	`---`
`timeout(double, value): Timeout`	`Timeout<T>(double, T): ITimeout<T>`
`suspend(): Event`	`---`
`event(): Event`	`Event<T>(): IEvent<T>`
`all_of(Event[]): AllOf`	`AllOf<T1,..>(T1,..): Condition<T1,..>`
`any_of(Event[]): AnyOf`	`AnyOf<T1,..>(T1,..): Condition<T1,..>`
`run(value): void`	`Run(), Run(double), Run(IEvent)`
`peek(): double`	`---`
`step(): void`	`---`

Tabella 4.2: Traduzione dei metodi e degli attributi di `Environment`.

`= env.Timeout(5);`", dove non sarà necessario specificare il tipo generico poiché viene implicitamente dedotto. Quindi, vedremo che le aggiunte, in generale, non rappresenteranno un ostacolo allo sviluppo, ma saranno soprattutto caratteristiche che miglioreranno l'integrazione su .NET.

4.4 Differenze rispetto a SimPy

Questa lunga sezione sarà dedicata alla descrizione delle varie discrepanze tra la nostra libreria e l'originale, e compreso il razionale che ha portato a fare le varie scelte. Ne approfitteremo per mostrare alcuni esempi d'uso di Dessert; il lettore ne potrà trovare molti altri in appendice B.

4.4.1 Funzionalità modificate

Inizieremo con il presentare le caratteristiche di SimPy che, per varie ragioni, abbiamo dovuto modificare.

4.4.1.1 Creazione delle entità della simulazione

Dessert, al fine di ottenere un design pulito e ordinato, espone unicamente delle interfacce[1], a differenza di SimPy, dove sono esposte direttamente e unicamente le classi[2]. A causa di ciò, gli elementi della simulazione devono essere

[1] In realtà, anche alcune classi sono pubbliche. Ciò è stato fatto per permettere la creazione di Armando, dato che IronPython richiede che esse siano così per poterle "studiare" a tempo di esecuzione. In ogni caso, abbiamo preso le dovute precauzioni per evitare che l'utente possa erroneamente istanziare tali classi.

[2] SimPy non avrebbe potuto fare altrimenti, dato che il concetto di interfaccia non è presente all'interno del linguaggio Python.

Elemento	SimPy	Dessert
Environment	`Environment()`	`Sim.NewEnvironment(...)`
Resource	`Resource(env, ...)`	`env.NewResource(...)`
Container	`Container(env, ...)`	`env.NewContainer(...)`
Store	`Store(env, ...)`	`env.NewStore(...)`
Condition	`Condition(env, ...)`	`env.Condition(...)`

Tabella 4.3: Differenze nei metodi di creazione degli elementi.

istanziati tramite delle *factory* [120], rappresentate dalla classe statica `Sim` e dall'interfaccia `IEnvironment`.

Per riassumere chiaramente come poter creare i vari elementi, abbiamo riportato in tabella 4.3 un confronto tra i metodi messi a disposizione da SimPy e quelli di Dessert. Per sinteticità, ne abbiamo inserito solo una parte, poiché quelli rimanenti non presentavano differenze significative.

4.4.1.2 Lettura dei valori restituiti dagli eventi

Quando abbiamo mostrato al lettore la libreria SimPy, abbiamo fatto notare il fatto che `yield`, in Python, sia una espressione e che ciò venga sfruttato per passare ai processi i valori restituiti dagli eventi; se `ev` fosse un evento, tramite lo statement "`val = yield ev`" ritroveremmo su `val` il valore prodotto da `ev`. Tale funzionalità, obiettivamente chiara ed elegante, non può essere replicata su alcuni linguaggi per .NET, dato che lì `yield` è solamente uno *statement*.

Pertanto, abbiamo dovuto trovare un meccanismo sostitutivo, decidendo che ogni istanza di `IEvent` avesse un attributo `Value`, sul quale fosse possibile recuperare il valore restituito da un evento alla propria attivazione. Tramite i generici e opportune ridefinizioni, inoltre, abbiamo cercato di far sì che, ove possibile, l'attributo `Value` non fosse semplicemente di tipo `object`, ma che avesse un tipo appropriato.

Nell'esempio riportato nel listato 4.7, mostriamo come la proprietà `Value` vada utilizzata *dopo* che l'evento atteso abbia avuto successo, in modo da essere sicuri che il valore sia realmente presente. Inoltre, l'esempio evidenzia il fatto che, se possibile, le interfacce di Dessert "memorizzano" il tipo del valore, al fine di sfruttare tutti i vantaggi del tipaggio statico.

4.4.1.3 Gestione del fallimento degli eventi

La gestione del fallimento degli eventi è uno dei punti *dolenti* del nostro progetto. Infatti, mentre in SimPy tale caratteristica è gestita con le eccezioni lanciabili dal metodo `throw` presente nei generatori, sui linguaggi considerati per .NET, sfortunatamente, non esistono metodi equivalenti né procedure o librerie esterne che offrano tale possibilità. Occorre anche ricordare che alcuni linguaggi per .NET hanno ulteriori limitazioni nell'uso dello statement `yield` all'interno di un blocco `try-catch`; ad esempio, in C# non è possibile porlo né nel blocco

Listato 4.7 Come recuperare su Dessert i valori restituiti dagli eventi.

```
Iterator Function Process(env As IEnvironment)
    As IEnumerable(Of IEvent)

    Dim t = env.Timeout(5, value:="A BORING EXAMPLE")
    Yield t
    ' t.Value è una stringa, perciò possiamo
    ' applicare i metodi di string su di esso.
    Console.WriteLine(t.Value.Substring(2, 6))
    Dim intStore = env.NewStore(Of Double)()
    intStore.Put(t.Delay)
    Dim getEv = intStore.Get()
    Yield getEv
    ' getEv.Value è un numero decimale, perciò lo
    ' possiamo moltiplicare per 2.5, come previsto.
    Console.WriteLine(getEv.Value * 2.5)
End Function

Sub Run()
    Dim env = Sim.NewEnvironment()
    env.Start(Process(env))
    env.Run()
End Sub
```

Listato 4.8 Gestione del fallimento di un evento.

```
1  IEnumerable<IEvent> EventFailer(IEvent<string> ev) {
2      ev.Fail("SOMETHING BAD HAPPENED");
3      yield break;
4  }
5
6  IEnumerable<IEvent> Process(IEnvironment env) {
7      var ev = env.Event<string>();
8      env.Start(EventFailer(ev));
9      yield return ev;
10     if (ev.Failed) Console.WriteLine(ev.Value);
11 }
12
13 void Run() {
14     var env = Sim.NewEnvironment();
15     env.Start(Process(env));
16     env.Run();
17 }
```

`try`, né in quello `catch` [121], mentre in VB.NET può essere posto soltanto nel blocco `Try` [122].

Perciò, non potendo replicare *esattamente* quella funzionalità, abbiamo introdotto dentro `IEvent` una proprietà booleana, `Failed`, che indichi se l'evento in questione sia fallito o meno. Inoltre, abbiamo rilassato il vincolo presente nel metodo `Fail`, facendo in modo che potesse accettare un valore qualunque e non strettamente un'eccezione, differenziandoci ulteriormente dalle scelte di SimPy. La proprietà `Failed` è, inizialmente, falsa; dopodiché, quando l'evento si è attivato, allora soltanto una proprietà, tra `Succeeded` e `Failed`, potrà essere vera: quindi, se `Succeeded` è vero, allora `Failed` sarà e rimarrà falso, e viceversa.

Nell'esempio, presente nel listato 4.8, mostriamo come verificare se un evento sia fallito. In particolare, il processo crea un evento, `ev`, che verrà passato a un altro processo, `EventFailer`, in modo che lo faccia fallire; successivamente, il processo principale, dopo essersi messo in attesa di `ev`, si riattiverà, scoprendo che l'evento in questione è fallito. Grazie all'introduzione dei generici, abbiamo che la proprietà `Value` di `ev` è opportunamente tipata; relativamente all'esempio, essa risulta essere una stringa.

Come è facile aspettarsi, l'output dell'esempio in 4.8 sarà semplicemente "`SOMETHING BAD HAPPENED`".

4.4.1.4 Gestione degli interrupt

Analogamente alla gestione del fallimento degli eventi, anche per la gestione degli interrupt abbiamo dovuto modificare il design di SimPy, in modo da poterlo adattare alle specifiche degli iteratori di .NET.

Listato 4.9 Gestione della ricezione di un interrupt.

```
1  IEnumerable<IEvent> Interrupter(IProcess victim) {
2      victim.Interrupt("NOW");
3      yield break;
4  }
5
6  IEnumerable<IEvent> Process(IEnvironment env) {
7      yield return env.Start(Interrupter(env.ActiveProcess));
8      object cause;
9      if (env.ActiveProcess.Interrupted(out cause))
10         Console.WriteLine("Interrupted at: " + cause);
11
12     // Le seguenti istruzioni sono commentate, poiché
13     // darebbero origine a un'eccezione. Infatti,
14     // il secondo segnale di interrupt non sarebbe catturato.
15     //
16     // yield return env.Start(Interrupter(env.ActiveProcess));
17     // yield return env.Timeout(5);
18 }
19
20 void Run() {
21     var env = Sim.NewEnvironment();
22     env.Start(Process(env), "INTERRUPTED");
23     env.Run();
24 }
```

Dato che un processo di Dessert non può ricevere le interruzioni sotto forma di eccezioni, abbiamo introdotto un nuovo metodo per `IProcess`, `Interrupted`, con il quale si possa indagare se un processo sia stato interrotto durante l'attesa di un evento. Inoltre, abbiamo aggiunto un overload per `Interrupted` che prenda per riferimento un oggetto e memorizzi su di esso il valore che può essere passato al metodo `Interrupt`. All'interno di SimPy quel valore è recuperabile dall'eccezione stessa.

Il listato 4.9 contiene un esempio di come gli interrupt possano essere gestiti all'interno di un processo Dessert, necessariamente in modo meno elegante e sicuro di quanto si possa fare su SimPy.

Il fatto che gli interrupt non siano più eccezioni rende più probabile dimenticarsi della loro possibile presenza; per evitarlo, abbiamo fatto in modo che, nel caso ci si dimentichi di controllarli ed effettivamente siano presenti degli interrupt, venga sollevata un'eccezione alla successiva `yield`.

Nell'esempio in 4.9, se le ultime due istruzioni non fossero commentate, otterremmo un'eccezione esattamente nel momento in cui il processo principale effettui l'attesa di un timeout, senza aver prima effettuato una chiamata a

`Interrupted`.

Dato che le interruzioni vengono anche usate dalle risorse con prelazione, abbiamo aggiunto a `IProcess` un altro metodo, `Preempted`, con il quale sia possibile controllare se un processo sia stato vittima di prelazione.

4.4.1.5 Creazione delle condizioni

Allo scopo di mantenere il più possibile l'informazione statica dei tipi degli eventi coinvolti in una condizione, abbiamo dovuto modificare il modo in cui le condizioni possono essere create e usate, differenziandoci, sfortunatamente anche in questo aspetto, da SimPy.

Ricordiamo che in SimPy le condizioni possono essere create fornendo:

- Una funzione di valutazione, che riceve una lista con tutti gli eventi della condizione e un dizionario contenente quelli avvenuti, insieme ai rispettivi valori.
- Una lista con tutti gli eventi che fanno parte della condizione.

Il fatto che gli eventi siano memorizzati in una lista impedisce di tenere traccia del loro tipo, perché, per forza di cose, se ne potrebbe al massimo ricordare il super tipo comune. Quindi, abbiamo dovuto scartare quel design e ne abbiamo dovuto progettare uno nuovo, basato sui generici.

In pratica, abbiamo creato una definizione generica di `ICondition`, dove a ogni parametro generico corrispondeva il tipo di un evento coinvolto nella condizione; dato che in C#, a differenza del C++ [123], non esistono modi per descrivere un tipo avente un numero indefinito di parametri generici, abbiamo creato delle interfacce che potessero esprimere condizioni con al massimo cinque eventi distinti.

Gli eventi che compongono la condizione sono esposti in proprietà denominate `Ev{N}`, dove `N` è la posizione dell'evento al momento della costruzione della condizione. Per la costruzione, si richiede di specificare gli eventi coinvolti e la funzione di valutazione, che, a differenza di SimPy, deve soltanto accettare come parametro un oggetto avente il tipo della condizione che si sta creando.

Per chiarire le idee, nel listato 4.10 mostriamo un semplice caso d'uso delle condizioni. Ne approfittiamo anche per far notare al lettore che i metodi `AllOf` e `AnyOf` si comportano in modo analogo agli equivalenti di SimPy, ad eccezione del fatto che, per le questioni tecniche descritte sopra, essi non possono accettare più di cinque eventi. Alla riga 12 possiamo osservare la costruzione di una condizione "personalizzata", dove specifichiamo, oltre agli eventi coinvolti, anche la funzione di valutazione; nello specifico, la abbiamo definita in modo che fosse uguale a quella usata dal metodo `AnyOf`.

Inoltre, aiutandoci con il listato 4.11, dove abbiamo inserito l'output dell'esempio, possiamo vedere altri concetti sempre legati alle condizioni: l'istruzione alla riga 6 stampa "`7`", poiché vengono attesi entrambi gli eventi di timeout; al contrario, in riga 17, viene stampato "`10`", poiché, per via della disgiunzione, la condizione risulta verificata appena sono passate 3 unità di tempo. Osserviamo

Listato 4.10 Utilizzo degli eventi condizione nei processi Dessert.

```
IEnumerable<IEvent> Process(IEnvironment env) {
    var t1 = env.Timeout(3);
    var t2 = env.Timeout(7);
    var c1 = env.AllOf(t1, t2);
    yield return c1;
    Console.WriteLine(env.Now);
    Console.WriteLine(c1.Value.ContainsKey(t1));
    Console.WriteLine(c1.Value.ContainsKey(t2));

    t1 = env.Timeout(3);
    t2 = env.Timeout(7);
    var c2 = env.Condition(
        t1, t2,
        c => c.Ev1.Succeeded || c.Ev2.Succeeded
    );
    yield return c2;
    Console.WriteLine(env.Now);
    Console.WriteLine(c2.Value.ContainsKey(t1));
    Console.WriteLine(c2.Value.ContainsKey(t2));
}

void Run() {
    var env = Sim.NewEnvironment();
    env.Start(Process(env));
    env.Run();
}
```

Listato 4.11 Output dell'esempio in 4.10.

```
1 7
2 True
3 True
4 True
5 10
6 False
7 False
8 True
```

anche che la proprietà `Value` è stata ridefinita, affinché restituisca un dizionario con gli eventi che sono accaduti *prima* dell'attivazione della condizione, associati ai rispettivi valori. Tale ridefinizione risulta necessaria per Armando, ma può risultare utile anche nelle simulazioni per Dessert. Occorre far notare che se un evento viene attivato *dopo* il successo della condizione, non comparirà nel dizionario restituito da `Value`, secondo le specifiche di SimPy.

Operatori `and` e `or` SimPy ridefinisce per gli eventi gli operatori di congiunzione e disgiunzione, al fine di avere un modo pratico e altamente leggibile con cui dichiarare le condizioni più comuni. Tuttavia, dato che Dessert espone soltanto delle interfacce, non avremmo potuto replicare tale utile funzionalità.

Come conseguenza di ciò, abbiamo definito altri metodi su `IEvent`[3], `And` e `Or`, che svolgessero un ruolo equivalente. Abbiamo inoltre cercato di far sì che, nel caso si cerchino di combinare due condizioni tramite quei metodi, il risultato sia una condizione composta dagli eventi delle due condizioni originali, e non composta dalle condizioni stesse. Ad esempio, supponendo di avere `ev1` di tipo `ICondition<T1, T2>` ed `ev2` di tipo `ICondition<T3, T4, T5>`, la loro congiunzione o disgiunzione avrà tipo `ICondition<T1, T2, T3, T4, T5>`, il che ne facilita notevolmente l'uso.

Se non avessimo preso le opportune precauzioni, la condizione risultante sarebbe stata, di fatto, una *condizione di condizioni*, cosa che avrebbe complicato l'accesso ai singoli eventi; infatti, se `cond` fosse il risultato della congiunzione o disgiunzione tra `ev1` e `ev2`, il secondo evento di `ev2` si sarebbe potuto accedere, in maniera contorta, tramite `cond.Ev3.Ev2`, mentre, grazie agli accorgimenti presi, esso risulta correttamente accessibile da `cond.Ev5`.

Un esempio di quanto abbiamo appena descritto è riportato nel listato 4.12, dove, precisamente alle righe 2 e 8, creiamo due eventi condizione, usando le "scorciatoie" fornite dai metodi `And` e `Or`. Osserviamo anche che, come auspicabile, il risultato delle combinazioni dei tre eventi è effettivamente una condizione composta da tre elementi (`Ev1`, `Ev2`, `Ev3`).

Per aiutare la comprensione dell'esempio, il listato 4.13 ne mostra l'output.

[3] Non esattamente su tale interfaccia, per via di questioni implementative che non descriveremo perché renderebbero questa parte decisamente tediosa.

Listato 4.12 Come gli operatori `and` e `or` sono esposti da Dessert.

```
IEnumerable<IEvent> Process(IEnvironment env) {
    var c1 = env.Timeout(3).And(env.Timeout(5)).And(env.Timeout(7));
    yield return c1;
    Console.WriteLine(env.Now);
    Console.WriteLine(c1.Value.ContainsKey(c1.Ev1));
    Console.WriteLine(c1.Value.ContainsKey(c1.Ev2));
    Console.WriteLine(c1.Value.ContainsKey(c1.Ev3));
    var c2 = env.Timeout(7).Or(env.Timeout(5).Or(env.Timeout(3)));
    yield return c2;
    Console.WriteLine(env.Now);
    Console.WriteLine(c2.Value.ContainsKey(c2.Ev1));
    Console.WriteLine(c2.Value.ContainsKey(c2.Ev2));
    Console.WriteLine(c2.Value.ContainsKey(c2.Ev3));
}

void Run() {
    var env = Sim.NewEnvironment();
    env.Start(Process(env));
    env.Run();
}
```

Listato 4.13 Output dell'esempio in 4.12.

```
7
True
True
True
10
False
False
True
```

Listato 4.14 Avvio dei processi ritardatari all'interno di Dessert.

```
1 IEnumerable<IEvent> Process(IEnvironment env, char id) {
2     Console.WriteLine("{0}: {1}", id, env.Now);
3     yield break;
4 }
5
6 void Run() {
7     var env = Sim.NewEnvironment();
8     env.StartDelayed(Process(env, 'A'), 7);
9     env.Start(Process(env, 'B'));
10    env.Run();
11 }
```

4.4.1.6 Avvio dei processi "ritardatari"

A differenza di SimPy, dove viene esposto un processo apposito, `start_delayed`, per avviare un altro processo con un dato ritardo abbiamo creato una variante di `Start`, `StartDelayed`, che accettasse come parametro il ritardo del processo da far partire.

Abbiamo fatto questa scelta sia per questioni estetiche, dato che riportando quella funzionalità in C# avremmo ottenuto codice meno elegante, sia per questioni di efficienza, visto che, con la nostra soluzione, si avvia un solo processo, non due. Un esempio d'uso è presentato nel listato 4.14.

4.4.2 Funzionalità aggiunte

Scrivendo esempi per questa relazione e test per provare la nostra implementazione di Dessert, abbiamo maturato una discreta padronanza dei costrutti e dei concetti di SimPy, identificando parti che, a nostro avviso, avrebbero beneficiato di aggiunte per facilitare lo sviluppo o per sfruttare peculiarità dell'ambiente .NET.

Quindi, presenteremo ora le nostre personali aggiunte alla libreria Dessert, vedremo in cosa tali aggiunte consistano, e illustreremo da quali necessità pratiche siano state motivate.

Le linee guida che abbiamo seguito nella progettazione delle aggiunte sono state, fondamentalmente, la semplicità d'uso del risultato finale e la necessità di non scostarsi eccessivamente da ciò che SimPy espone.

4.4.2.1 Uso dei generici nelle interfacce

Come è già trapelato da alcuni esempi fatti in precedenza, abbiamo parametrizzato le nostre interfacce, in modo che l'informazione di tipo statico fosse propagata il più possibile, coinvolgendo anche elementi complessi come le condizioni, le callback e le risorse.

In particolare, abbiamo agito sui seguenti aspetti:

- Ove possibile, abbiamo cercato di *tipare* il valore corrispondente agli eventi. Ciò è stato fattibile con `ITimeout`, `IEvent`[4] e `ICondition`, mentre non ci è stato possibile agire su elementi come i processi e gli interrupt, perché lì non vi sarebbe stato modo di gestirne il tipo, senza usare cast a tempo di esecuzione.
- Le condizioni, e le relative operazioni di congiunzione/disgiunzione, cercano di preservare il tipo degli elementi coinvolti.
- Le callback associabili agli eventi e le funzioni di valutazione delle condizioni ricevono come parametro un evento avente un tipo specifico, e non un generico `IEvent`.
- Le interfacce `IStore` e `IFilterStore` sono generiche, al fine di mantenere un controllo sul tipo degli elementi memorizzati in esse.
- Anche gli eventi di tipo *call*, di cui non abbiamo ancora parlato, sono generici. A differenza degli altri punti dove abbiamo agito, questi richiedono minimi controlli a tempo di esecuzione.

4.4.2.2 Generatore di numeri casuali per `IEnvironment`

Considerato che la maggior parte delle simulazioni usa un generatore di numeri casuali per rendere realistiche le meccaniche e le interazioni dei processi, abbiamo deciso di aggiungerne uno dentro le istanze di `IEnvironment`. Essendo che i processi hanno assai frequentemente un riferimento a un'istanza di tale classe, abbiamo ritenuto che quello fosse il posto migliore dove collocare la nostra aggiunta.

Il generatore di numeri casuali è disponibile tramite la proprietà `Random`, a cui corrisponde un'istanza della classe `TRandom`, configurata in modo da utilizzare un generatore di tipo Mersenne twister [116].

4.4.2.3 Politica delle code nelle risorse

Dessert offre quattro tipi diversi di code da utilizzare all'interno delle risorse. Di base, esse seguono la politica FIFO, ma non è detto che essa sia adatta alla situazione che stiamo cercando di modellare.

Infatti, è plausibile che gli oggetti depositati in un magazzino siano recuperati in ordine inverso di arrivo, oppure può anche accadere che, nella manipolazione della coda, l'ordine di arrivo sia del tutto irrilevante e che gli elementi siano estratti in modo casuale.

[4] La libreria Dessert contiene ben tre definizioni di `IEvent`, che differiscono per il numero dei parametri di tipo e per il loro uso. La definizione senza parametri è estesa da ogni altro evento, in modo che possa essere usata come parametro generico per l'interfaccia `IEnumerable`, usata da ogni processo. La versione con due parametri è usata per scopi interni, mentre la dichiarazione con un solo parametro rappresenta quegli eventi che si comportano come le istanze di `Event` in SimPy.

Operazione\Tipo coda	`Fifo`	`Lifo`	`Priority`	`Random`
`IResource.Request`	✓	✓	✓	✓
`IResource.Release`	✓	✗	✗	✗
`IPreemptiveResource.Request`	✗	✗	✓	✗
`IPreemptiveResource.Release`	✓	✗	✗	✗
`IContainer.Get`	✓	✓	✓	✓
`IContainer.Put`	✓	✓	✓	✓
`IStore.Get`	✓	✓	✓	✓
`IStore.Put`	✓	✓	✓	✓
`IFilterStore.Get`	≈	≈	≈	≈
`IFilterStore.Put`	✓	✓	✓	✓

Tabella 4.4: Tipi di code applicabili alle varie risorse.

Quindi, abbiamo definito delle nuove modalità di funzionamento delle code, codificate dall'enumerazione `WaitPolicy`, che possono essere specificate nel momento in cui si costruisce una risorsa. Le modalità a disposizione sono:

- `FIFO`, estrazione nell'ordine di arrivo.
- `LIFO`, estrazione in ordine inverso di arrivo.
- `Priority`, estrazione in base alla priorità assegnata.
- `Random`, estrazione in modo casuale.

La tabella 4.4 riassume sinteticamente quali tipi di code possano essere associati alle operazioni sulle risorse; anche se nella tabella non è presente, occorre sottolineare il fatto che, sia per `IStore` sia per `IFilterStore`, è possibile specificare la coda da usare per gli oggetti (qualunque tipo è valido).

In merito a `IFilterStore`, la tabella riporta un simbolo di approssimazione per la coda di recupero degli elementi. Ciò è dovuto al fatto che, come avevamo spiegato nel capitolo 2, l'ordine della coda di recupero viene *generalmente* seguito ma, nel caso le funzioni di filtro dei primi processi rifiutino degli elementi, ci potrebbero essere degli avanzamenti di processi più indietro nella coda.

Infine, nei listati 4.15 e 4.16 mostriamo un esempio d'uso delle nuove politiche e un possibile output dello script. Nell'esempio creiamo uno *store* avente le politiche di base per le code di prelievo e inserimento, mentre per la coda degli oggetti si usa una politica casuale. Quindi, leggendo l'output, si può osservare che i processi vengono correttamente serviti nell'ordine in cui sono arrivati, ma gli oggetti, nonostante siano stati memorizzati ordinatamente, vengono distribuiti in modo casuale.

4.4.2.4 Evento per richiamare sotto procedure

Un problema che si nota soltanto scrivendo simulazioni piuttosto complesse è dato dal fatto che, a causa dell'uso degli iteratori, non è possibile "richiamare"

Listato 4.15 Specifica della politica delle code.

```
IEnumerable<IEvent> Getter(IStore<int> store, char id) {
    var getEv = store.Get();
    yield return getEv;
    Console.WriteLine("{0}: {1}", id, getEv.Value);
}

void Run() {
    var env = Sim.NewEnvironment(seed: 21);
    const int capacity = 10;
    // Gli oggetti vengono conservati in ordine "casuale".
    var store = env.NewStore<int>(capacity, WaitPolicy.FIFO,
                                  WaitPolicy.FIFO,
                                  WaitPolicy.Random);
    for (var i = 0; i < capacity; ++i)
        store.Put(i);
    for (var i = 0; i < capacity; ++i)
        env.Start(Getter(store, (char) ('A' + i)));
    env.Run();
}
```

Listato 4.16 Output dell'esempio in 4.15.

```
A: 6
B: 5
C: 0
D: 8
E: 2
F: 3
G: 9
H: 4
I: 1
J: 7
```

sotto procedure che restituiscano valori nell'iteratore principale; quindi, il codice dei processi tende a diventare piuttosto esteso e poco modulare.

Python, dalla versione 3.3 in poi, pone rimedio al problema aggiungendo al linguaggio il costrutto `yield from`, con il quale è possibile delegare la generazione a un secondo iteratore [124]. Tale funzionalità è vagamente replicabile in .NET tramite un ciclo `for`, ma così facendo otterremmo un risultato non ottimale dal punto di vista della pulizia del codice, dato che sarebbe poco intuitivo usare un'iterazione per realizzare, implicitamente, una chiamata di funzione.

Sarebbe anche possibile far coincidere la sotto procedura con un sotto processo, ma ciò, per quanto funzionante e chiaro, sarebbe poco efficiente a livello di consumo di risorse. Infatti, allocare e gestire un processo richiede risorse di calcolo e di memoria, le quali potrebbero diventare eccessive in una simulazione su larga scala.

Così, abbiamo introdotto il concetto di evento *call*, che consenta di richiamare sotto processi come se fossero delle sotto procedure, di fatto evitando i consumi aggiuntivi in termini di memoria. Una funzione *invocabile* è, superficialmente, identica a un processo, poiché essa può restituire eventi di ogni tipo e può opzionalmente terminare la propria esecuzione restituendo un dato valore; tuttavia, nel momento in cui viene invocata, il suo codice sostituisce momentaneamente quello del processo chiamante, in modo tale che gli eventi sembrino restituiti da lui.

Una funzione di quel tipo può, ovviamente, effettuare altre chiamate al suo interno, permettendo un uso arbitrario della ricorsione. A tal proposito, il listato 4.17 presenta una famosa applicazione della ricorsione, il calcolo dei numeri di Fibonacci, al nostro sistema di chiamate di funzione.

Un processo produttore di numeri di Fibonacci richiama, tramite l'evento restituito da `env.Call`, la sotto procedura `FibonacciFunc`. Essa applica il classico algoritmo ricorsivo, sfruttando ulteriori chiamate a quel metodo di `IEnvironment`; una volta che il risultato sarà pronto, lo restituirà al chiamante tramite l'invocazione di `env.Exit`[5]. Il produttore troverà il numero richiesto nel campo `Value` dell'oggetto *call* ricevuto con la prima chiamata, e potrà memorizzarlo nello *store* dedicato.

4.4.2.5 Nomi per eventi e risorse

Per alcuni eventi, e per tutte le risorse, abbiamo aggiunto la possibilità di specificare un nome. Tale nome, memorizzato nella proprietà `Name`, può tornare utile nelle stampe e nel debugging, dato che le implementazioni di `ToString` lo usano per produrre la stringa restituita.

[5] Abbiamo scelto di mantenere l'uso della funzione `Exit`, invece di introdurne una con un nome più consono, come `Return`, per non aggiungere un carico cognitivo per l'utilizzatore. Una volta compreso il fatto che, dopo aver richiamato il metodo `Exit`, il processo viene interrotto e si restituisce il valore dato, si potrà applicare il medesimo concetto anche alle chiamate di funzione effettuate tramite `Call`.

Listato 4.17 Eventi *call* applicati a un produttore di numeri di Fibonacci.

```
1  IEnumerable<IEvent> FibonacciFunc(IEnvironment env, int n) {
2      if (n <= 0) yield return env.Exit(0);
3      else if (n == 1) yield return env.Exit(1);
4      else {
5          var call = env.Call<int>(FibonacciFunc(env, n - 1));
6          yield return call;
7          var n1 = call.Value;
8          call = env.Call<int>(FibonacciFunc(env, n - 2));
9          yield return call;
10         var n2 = call.Value;
11         yield return env.Exit(n1 + n2);
12     }
13 }
14
15 IEnumerable<IEvent> Producer(IEnvironment env,
16                              IStore<int> store, int n)
17 {
18     var call = env.Call<int>(FibonacciFunc(env, n));
19     yield return call;
20     store.Put(call.Value);
21 }
22
23 IEnumerable<IEvent> Consumer(IStore<int> store) {
24     var getEv = store.Get();
25     yield return getEv;
26     Console.WriteLine(getEv.Value);
27 }
28
29 void Run() {
30     const int count = 10;
31     var env = Sim.NewEnvironment();
32     var store = env.NewStore<int>();
33     for (var i = 0; i < count; ++i) {
34         env.Start(Producer(env, store, i));
35         env.Start(Consumer(store));
36     }
37     env.Run();
38 }
```

Listato 4.18 Costruzione di eventi aventi un nome.

```
IEnumerable<IEvent> Process(IEnvironment env) {
    var a = env.NamedTimeout(3, "A");
    var b = env.NamedTimeout(5, "B");
    var c = env.NamedTimeout(7, "C");
    var cond = a.And(b.Or(c));
    yield return cond;
    Console.WriteLine(env.Now);
    foreach (var e in cond.Value.Keys)
        Console.WriteLine(e.Name);
}

void Run() {
    var env = Sim.NewEnvironment();
    env.Start(Process(env));
    env.Run();
}
```

Il listato 4.18 presenta la creazione di tre timeout, a ciascuno dei quali viene assegnato un nome; successivamente, il nome viene recuperato per ottenere una stampa più dettagliata.

4.4.2.6 Strumenti per raccolta statistiche

La versione 2.3 di SimPy possedeva tutta una serie di strumenti per la registrazione e la raccolta dei dati [125], ma, per questioni di tempo, essi non stati migrati alla versione 3 [126]. In particolare, gli sviluppatori di SimPy stanno prendendo tempo per pensare un nuovo design per tali strumenti: se tutto procede come gli sviluppatori sperano, il nuovo sistema sarà pronto per la versione 3.1 [126].

Tuttavia, noi ci siamo trovati in una situazione poco piacevole, dato che avremmo dovuto implementare gli strumenti per le registrazioni, utilissimi nella raccolta di statistiche durante una simulazione, ma senza avere un'interfaccia di riferimento. Così, abbiamo deciso di basarci sulle interfacce esposte dalla versione 2.3, che hanno fornito un'ottima base di partenza.

Nel dettaglio, esse consistono di due entità: il *monitor* e il *tally*. Entrambe sono adibite alla raccolta di dati numerici, tramite il metodo `Observe`, e alla produzione di statistiche, attraverso metodi come `Mean`; tuttavia, esse seguono un approccio decisamente diverso per realizzare i loro obiettivi.

I registratori di tipo *tally* non memorizzano tutte le osservazioni, ma aggiornano, a ogni chiamata di `Observe`, alcuni valori aggregati che saranno utili per produrre le statistiche: ad esempio, verrà aggiornata la somma, il conteggio delle osservazioni, etc etc. Al contrario, i *monitor* memorizzano ogni singola osservazione, producendo una lista, `Samples`, liberamente consultabile da par-

Listato 4.19 Semplice uso degli strumenti per la raccolta di statistiche.

```
IEnumerable<IEvent> Person(IEnvironment env,
                           IResource queue, ITally rec)
{
    using (var req = queue.Request()) {
        var startTime = env.Now;
        yield return req;
        rec.Observe(env.Now - startTime);
        var workTime = env.Random.Next(3, 10);
        yield return env.Timeout(workTime);
    }
}

IEnumerable<IEvent> Spawner(IEnvironment env, ITally rec) {
    var queue = env.NewResource(2);
    while (true) {
        env.Start(Person(env, queue, rec));
        yield return env.Timeout(env.Random.Next(2, 5));
    }
}

void Run() {
    var env = Sim.NewEnvironment(seed: 42);
    var rec = env.NewTally();
    env.Start(Spawner(env, rec));
    env.Run(2*60);
    Console.WriteLine("Totale clienti: {0}", rec.Total());
    Console.WriteLine("Attesa media: {0:.0}", rec.Mean());
}
```

te dell'utente; le statistiche vengono calcolate *al volo* a partire proprio da tale insieme di dati.

Il listato 4.19 contiene un classico esempio d'uso dei registratori di dati, cioè, misurare il tempo medio di attesa in una coda. Abbiamo già fatto una cosa equivalente nell'esempio della banca, in sezione 2.6, ma, visto che SimPy 3 non è provvisto degli strumenti per la registrazione, avevamo adottato una strategia manuale per memorizzare e analizzare quei dati.

Nell'esempio, possiamo notare che viene creato un registratore di tipo *tally*, usando il metodo `NewTally` esposto da `IEnvironment`; dopodiché, l'istanza del registratore viene passata ai processi, i quali registreranno il tempo di attesa richiesto per ottenere il controllo di una data risorsa. Infine, a simulazione conclusa, accediamo ad alcune statistiche del registratore per osservare i dati prodotti dal nostro esperimento.

Un registratore può essere associato a un dato ambiente, se creato tramite `IEnvironment`, oppure no, se lo si crea attraverso la classe statica `Sim`. La differenza tra le due scelte la si può notare solo sapendo che, per ogni osservazione, si tiene implicitamente conto del tempo, al fine di rendere disponibili statistiche come media e varianza pesate nel tempo: se un registratore è associato a un ambiente, si sfrutterà il valore restituito dalla proprietà `Now` per ottenere il tempo da associare ai dati, mentre se non è associato a nulla, il tempo associato sarà sempre zero.

La nostra implementazione, per quanto funzionante, è stata realizzata solo per colmare un vuoto che, per questioni di praticità e di completezza, non ci saremmo potuti permettere. Per cui, non è da escludere che, dopo che gli sviluppatori di SimPy avranno stabilito il nuovo design, Dessert scarti la propria implementazione e adegui la parte di registrazione a quella del futuro SimPy.

4.4.3 Funzionalità rimosse

In generale, abbiamo cercato di replicare ogni singola funzionalità di SimPy. Tuttavia, ve ne sono un paio, che, per questioni di tempo, non abbiamo potuto realizzare. Faremo ora un breve cenno a tali caratteristiche, le quali potrebbero essere implementate in lavori futuri.

4.4.3.1 Simulazione in tempo reale

SimPy, all'interno del modulo `simpy.rt`, offre gli strumenti necessari per realizzare simulazioni dove il tempo scorra in modo *reale* e non *virtuale*, come nelle simulazioni descritte finora. Infatti, spesso annotiamo i tempi indicati nel codice con dei commenti, ma sappiamo che la simulazione scorrerà il più velocemente possibile, facendo avanzare il tempo nel modo indicato in sezione 1.2.2.

Frequentemente, ciò è il comportamento desiderato, perché si cercano statistiche sul modello implementato, e non delle osservazioni su come esso procederebbe se i tempi indicati nei *timeout* fossero reali.

Ad ogni modo, SimPy offre anche tale possibilità. Nel modulo `simpy.rt` è presente uno speciale ambiente, `RealtimeEnvironment`, il cui tempo di esecuzione è *proporzionale* ai tempi specificati nelle varie timeout. Di base, tale classe associa un'unità di tempo virtuale a un secondo reale, così che un timeout di 7 unità, ad esempio, impieghi 7 secondi prima di attivarsi. Viene anche offerta la possibilità di cambiare la scala di conversione e la possibilità di ricevere opportuni errori, quando i tempi che si dovrebbero attendere sono inferiori a quelli necessari al motore di simulazione per la sola elaborazione degli eventi.

4.4.3.2 Avanzamento "manuale" della simulazione

Ciascun tipo di ambiente esposto da SimPy possiede un metodo, `step`, con il quale è possibile far avanzare procedere la simulazione *manualmente*. Di fatto, il metodo `run` non fa altro che richiamare `step`, finché non vi sono più elementi da processare.

Listato 4.20 Preparazione della simulazione di una banca, tradotto da 2.30.

```
1 open Dessert
2 open MoreLinq // Espone MinBy, usato dentro Spawner
3
4 Sim.CurrentTimeUnit <- TimeUnit.Minute
5 let avgIncomingT, avgServiceT = (3).Minutes(), (10).Minutes()
6 let queueCount = 3 // Numero sportelli
7 let bankCap, bankLvl = 20000.0, 2000.0 // Euro
8 let waitTally, servTally = Sim.NewTally(), Sim.NewTally()
9 let mutable totClients = 0
```

Non avendo trovato un'immediata utilità o una particolare necessità di avere quella funzione, la abbiamo scartata dalla nostra implementazione.

4.5 Esempio della banca

A conclusione del capitolo 2, precisamente in sezione 2.6, avevamo preparato una simulazione discretamente complessa, con la quale mostrare, in un colpo solo, molte delle funzionalità che avevamo visto per SimPy. Faremo ora una cosa analoga anche per Dessert, sia al fine di compiere una panoramica di quanto descritto nel capitolo, sia per mostrare, con un esempio concreto, come la nostra libreria si rapporti all'originale.

Le specifiche della nostra simulazione saranno le stesse che sono state presentate per l'esempio di SimPy; di fatto, ciò che stiamo per eseguire è una *traduzione* nel nostro linguaggio del precedente esempio.

Questa parte è organizzata secondo la stessa struttura impiegata in sezione 2.6, in modo da facilitare il confronto tra le varie parti della simulazione. Sempre al fine di rendere più semplice la comparazione, ciascun listato riporterà quale sia l'originale da cui è stato tradotto.

Il codice completo dell'esempio è riportato in appendice A.

4.5.1 Variabili richieste e accessorie

La nostra "preparazione", presentata nel listato 4.20, è significativamente diversa da quella per SimPy. Innanzitutto, non dobbiamo dichiarare un generatore di numeri casuali, poiché esso è già esposto dall'istanza di `IEnvironment`, la quale dovrà essere necessariamente creata per far partire la simulazione. La creazione di tale istanza è rimandata al listato 4.23, poiché abbiamo deciso di inserire in un'apposita funzione il codice adibito all'impostazione e all'esecuzione della simulazione.

Si può anche notare che, per esprimere le tempistiche della simulazione, lo script faccia uso di appositi *extension method* [127] definiti da Dessert. Essi, a nostro avviso, semplificano sia la lettura del codice, dato che l'unità di misura

Listato 4.21 Definizione del processo relativo ai clienti, tradotto da 2.31.

```
let client(env: IEnvironment, queue: IResource,
           bank: IContainer, amount, get) = seq<IEvent> {
    use req = queue.Request()
    let s1 = env.Now
    yield upcast req
    waitTally.Observe(env.Now - s1)
    let s2 = env.Now
    yield upcast env.Timeout(env.Random.Exponential(1.0/avgServiceT))
    if get then yield upcast bank.Get(amount)
    else yield upcast bank.Put(amount)
    servTally.Observe(env.Now - s2)
}
```

è esplicitata dal metodo usato, sia la sua scrittura, visto che si prendono carico di eseguire le dovute conversioni, cosa che, spesso, causa notevoli "grattacapi" agli sviluppatori.

Infine, il listato in 4.20 sfrutta gli strumenti per la registrazione dei dati che abbiamo implementato per Dessert, al fine di raccogliere le statistiche tu tempi di attesa e di servizio. In particolare, vengono usati due *tally*, considerato che abbiamo bisogno soltanto dei risultati, le medie, e non dei singoli dati; se avessimo avuto bisogno di tutti i dati, invece, avremmo dovuto istanziare due *monitor*.

4.5.2 Il processo cliente

Il processo relativo ai clienti della banca, mostrato nel listato 4.21, non si scosta molto dall'originale in Python. Escluse le differenze prettamente sintattiche e tecniche dovute all'uso del linguaggio F#, come l'uso di `upcast` a ogni `yield`[6], gli unici cambiamenti degni di nota sono dati dall'uso dei due *tally* per registrare i tempi di attesa.

Quindi, come auspicabile, il passaggio a Dessert non ha prodotto notevoli cambiamenti nel codice del processo cliente.

4.5.3 Il processo generatore di clienti

Fortunatamente, la traduzione in Dessert del generatore di clienti, visibile nel listato 4.22, è praticamente identica a ciò che avevamo scritto per SimPy. Ciò è un ottimo risultato, perché dimostra che uno dei nostri obiettivi principali, la compatibilità *concettuale* con SimPy, è stato raggiunto.

[6] Il linguaggio F# non effettua in automatico gli upcast [128], il che rende necessario l'uso dell'omonimo comando per far sì che le istanze restituite da `IEnvironment` siano riconosciute come `IEvent`.

Listato 4.22 Definizione del processo generatore di clienti, tradotto da 2.32.

```
let rec spawner(env: IEnvironment, queues: IResource list,
                bank) = seq<IEvent> {
    yield upcast env.Timeout(env.Random.Exponential(1.0/avgIncomingT))
    let queue = queues.MinBy(fun q -> q.Count)
    let amount = float(env.Random.Next(50, 500))
    let get = env.Random.NextDouble() < 0.4
    env.Start(client(env, queue, bank, amount, get)) |> ignore
    totClients <- totClients + 1
    // Richiama ricorsivamente la funzione spawner.
    // Grazie all'ottimizzazione della ricorsione in coda,
    // questo equivale all'uso di un costrutto while.
    yield! spawner(env, queues, bank)
}
```

4.5.4 Avvio della simulazione e raccolta dati

Anche per il listato 4.23, dove inizializziamo e avviamo la simulazione, non vi è molto da dire, poiché il codice è assai simile a quello in Python; il fatto che F# presenti costrutti analoghi a tale linguaggio, come le *list comprehension* [129], aiuta la nostra traduzione a rimanere piuttosto aderente all'originale.

Escluse le inevitabili differenze causate dal linguaggio, possiamo soltanto notare l'uso dei due *tally* al fine di ottenere le medie per i tempi di attesa. Come già detto, il fatto che non vi siano differenze importanti è, dal nostro punto di vista, un eccellente risultato.

4.5.5 Analisi dei dati

Come prevedibile, i dati che possono essere ottenuti dalla nuova simulazione per Dessert sono "compatibili" con quanto abbiamo riscontrato con SimPy. Perciò, non abbiamo nulla da aggiungere rispetto a quanto detto in sezione 2.6.6.

Listato 4.23 Preparazione della simulazione di una banca, tradotto da 2.33.

```
let run() =
    let env = Sim.NewEnvironment(seed = 21)
    let queues = [for x in 1..queueCount do
                      yield env.NewResource(1)]
    let bank = env.NewContainer(bankCap, bankLvl)

    // Avvio della simulazione
    env.Start(spawner(env, queues, bank)) |> ignore
    env.Run(until = (5).Hours())

    // Raccolta dati statistici
    let lvl = bank.Level
    printfn "Finanze totali al tempo %.2f: %g" env.Now lvl
    printfn "Clienti entrati: %d" totClients
    printfn "Clienti serviti: %d" servTally.Count
    printfn "Tempo medio di attesa: %.2f" (waitTally.Mean())
    printfn "Tempo medio di servizio: %.2f" (servTally.Mean())
```

Capitolo 5

Prestazioni e confronti

Per quanto noi ritenessimo che, scrivendo la nostra libreria in un linguaggio compilato e fortemente tipato, le prestazioni a tempo di esecuzione sarebbero state sensibilmente superiori, abbiamo voluto validare le nostre aspettative con opportuni test e verifiche.

In generale, i test sono serviti sia per migliorare il nostro progetto, scoprendo quali fossero le debolezze e i colli di bottiglia, sia per avere la certezza che i nostri obiettivi, esclusivamente in merito alle prestazioni, fossero realmente raggiunti.

Come si suole fare in ambito scientifico, scopo di questo capitolo sarà quello di dettagliare quali siano state le prove eseguite, su quale macchina siano state portate a termine e quali siano i parametri di analisi di ciascun confronto. Infatti, nei nostri confronti non ci limiteremo ai sistemi Windows, ma valuteremo anche le prestazioni in ambito Linux. Ciò sarà fatto sia per dimostrare che il nostro lavoro funziona anche in altri ambienti di esecuzione, sia per scoprire se Dessert possa mantenere le proprie prestazioni anche in ambienti non esattamente favorevoli per il codice .NET.

Cominceremo con il compiere, in sezione 5.1, una panoramica dei confronti che eseguiremo e, sempre in tale sezione, descriveremo le specifiche tecniche della macchina sulla quale abbiamo eseguito i test. Dopodiché, dalla sezione 5.2 alla 5.4, discuteremo più dettagliatamente di ciascuna prova e ne analizzeremo criticamente i risultati.

5.1 Panoramica dei confronti compiuti

Nella scelta di cosa analizzare con i test, abbiamo voluto indagare sia sulle prestazioni del *cuore* del simulatore, effettuando prove mirate che ne mettessero alla prova l'efficienza, sia sulla velocità complessiva di una simulazione di medie dimensioni. Il secondo tipo di test, a differenza del primo, non coinvolge soltanto il simulatore, ma anche il codice che compone la simulazione; infatti, mentre nei test mirati vedremo che i processi sono piuttosto semplici e brevi, nel test

concreto i processi saranno necessariamente più articolati, il che comporterà un'equa ripartizione del tempo di calcolo tra il simulatore e i processi stessi.

In totale, abbiamo progettato tre confronti, tra i quali due erano di tipo *tecnico* e uno riguardava l'esecuzione di una simulazione reale. I test tecnici, descritti nelle sezioni 5.2 e 5.3, consistevano nella definizione di processi il cui unico scopo era quello di restituire un certo tipo di evento, al fine di monitorare la velocità di elaborazione del motore di simulazione; la simulazione reale, invece, riguardava l'analisi del comportamento di una rete complessa e ci ha permesso di osservare il comportamento dei vari motori in una casistica reale.

Nelle nostre prove abbiamo messo a confronto la libreria Dessert, la libreria SimPy e Armando, il layer di traduzione verso le simulazioni scritte in Python. Quindi, vediamo quali piattaforme software siano state prese in considerazione per eseguire i diversi test e quali siano le capacità della macchina su cui tutto è stato portato a termine.

5.1.1 Piattaforme software utilizzate

Partendo dai sistemi operativi impiegati, abbiamo scelto di eseguire i confronti su Microsoft Windows 8, versione a 64 bit, e Ubuntu Linux 12.04, versione a 32 bit; entrambi i sistemi operativi riportavano tutti gli aggiornamenti stabili rilasciati prima del 20 Settembre 2013 e non presentavano modifiche particolari al kernel o al software di base.

Su Windows abbiamo eseguito il codice .NET sul framework di Microsoft, versione 4.5, mentre su Ubuntu abbiamo optato per la versione 2.10 della piattaforma Mono.

Per quanto riguarda Python, invece, sia su Windows sia su Ubuntu abbiamo provato il codice sulla versione 2.7 di tale linguaggio; l'interprete è sempre stato eseguito con il flag `-OO`, che forza l'applicazione del massimo livello di ottimizzazione.

5.1.2 Specifiche tecniche della macchina di test

Le simulazioni sono dei processi fortemente *CPU-bound* [130] perché sfruttano unicamente la potenza di calcolo e la memoria principale. A meno che il codice della simulazione non acceda esplicitamente alla memoria secondaria, il motore del simulatore non ha, per il momento, bisogno di farlo. Dunque, nel dettagliare le caratteristiche della macchina su cui i test sono stati eseguiti, sarà necessario prendere in considerazione soltanto il processore e la quantità di memoria principale a disposizione.

Per quanta riguarda il processore, esso è un "vecchio" Intel Core 2 Duo, modello E4700 [131], avente due *core* con una frequenza di clock di 2.6 GHz e una memoria cache di secondo livello da 2 MB.

La memoria principale, invece, è costituita da due blocchi Corsair da 1 GB [132], per un totale di 2 GB a una frequenza di 800 MHz.

Listato 5.1 Definizione del processo usato nel primo confronto.

```
1 def timeoutBenchmark_PEM(env, counter):
2     while True:
3         yield env.timeout(counter.randomDelay())
4         counter.increment()
```

5.2 Primo confronto: generazione di timeout

Nel nostro primo esperimento abbiamo optato per verificare la capacità di gestire un gran numero di eventi e di processi da parte del simulatore. L'idea era quella di misurare, se possibile, la "velocità" di elaborazione del motore di simulazione; ciò implicava la scelta di processi che fossero *minimali*, in modo che il tempo di calcolo consumato al loro interno fosse sostanzialmente nullo, se confrontato con il tempo usato dal motore.

In base a tali requisiti, abbiamo stabilito di far eseguire un numero fissato, e crescente a ogni test, di processi aventi la definizione riportata nel listato 5.1. Possiamo notare che si tratta semplicemente di un processo che non termina mai e che restituisce soltanto dei timeout con intervalli casuali; se si immagina di eseguire un certo numero di quei processi, si capisce che il sistema viene messo notevolmente sotto sforzo, poiché si trova a gestire un numero cospicuo di processi e un mediamente equivalente numero di eventi.

Il corpo del generatore include anche opportune istruzioni utili per registrare i dati necessari alla misurazione della velocità del motore. Ogni processo, dopo aver atteso con successo un certo timeout, incrementa un contatore globale, con il quale sarà possibile avere il numero degli eventi complessivamente elaborati dal simulatore. Al fine di ottenere la velocità di elaborazione, quel conteggio viene diviso per il tempo di esecuzione della simulazione, misurato dall'avvio dei processi sino al termine della simulazione.

Inoltre, abbiamo cercato di misurare anche il consumo di memoria principale, con lo scopo di comprendere quanto il sistema fosse scalabile. Per fare ciò, abbiamo definito un ulteriore processo, mostrato nel listato 5.2, che si occupasse di campionare, a intervalli regolari, la quantità di memoria consumata dal processo di test. Effettuare il rilevamento della memoria non si è rivelato un problema per il codice .NET, mentre per Python abbiamo riscontrato alcune problematiche, dovute alla mancanza di un'interfaccia multi piattaforma per l'accesso alle risorse del sistema operativo. Come conseguenza, il rilevamento del consumo di memoria da parte di Python su Windows non sarà presente nei dati, mentre lo sarà quello su Linux.

5.2.1 Dati di partenza

Useremo un numero crescente di processi, a partire da 500 fino ad arrivare a 25000; non saliremo seguendo un incremento preciso, in quanto l'intervallo aumenterà al crescere del numero di processi.

Listato 5.2 Definizione del processo usato per campionare la memoria.

```
let memRec(env: IEnvironment, tally: ITally) = seq<IEvent> {
    while true do
        yield upcast env.Timeout(memRecFreq)
        let procMem = currProc.WorkingSet64
        let gcMem = GC.GetTotalMemory(false)
        tally.Observe(float((procMem+gcMem)/(1024L*1024L)))
}
```

Per ogni numero di processi ricaveremo i seguenti dati:

- Il numero medio di eventi elaborato ogni secondo, ottenuto dividendo il totale degli eventi elaborati per la durata della simulazione.
- Il consumo medio della memoria, ottenuto campionando a intervalli regolari la quantità di memoria principale usata dal processo.

Eseguiremo il tutto cinque volte, in modo che da avere dei dati leggermente più affidabili. I risultati esposti in seguito saranno la media di quanto si è ottenuto nelle cinque esecuzioni.

Ciascuna simulazione durerà mille unità di tempo e gli intervalli dei timeout potranno variare tra dieci e cinquanta unità di tempo.

5.2.2 Risultati ottenuti

Prima di discutere realmente sui risultati che sono stati ottenuti dal confronto, vorremmo che il lettore osservasse la figura 5.1 e notasse che ogni curva, indipendentemente dalla libreria e dal sistema operativo, segue un andamento pseudo logaritmico. Questo è un risultato eccellente sia per Dessert, sia per SimPy, poiché ciò indica che il motore, di per se, è potenzialmente scalabile e può arrivare a gestire un gran numero di processi, senza che le prestazioni precipitino troppo rapidamente.

Il fatto che le curve seguano un andamento logaritmico è dovuto a una corretta implementazione degli heap sottostanti i motori; infatti, una simulazione, dal punto di vista del motore, non è altro che una sequenza di operazioni sullo heap. Considerato che, su tale struttura dati, le operazioni hanno *al massimo* complessità logaritmica, risulta chiaro perché il grafico riportato abbia tale andamento.

Osservando la figura si nota immediatamente che Dessert su Windows, linea verde, spicca sulle altre curve, in quanto offre prestazioni decisamente superiori; appena sotto abbiamo Armando, sempre su Windows, e di nuovo Dessert, ma stavolta su Ubuntu. In questo test SimPy non ha brillato, né su Windows né su Ubuntu.

Da tale risultato possiamo evincere che il motore di Dessert sembra essere piuttosto efficiente, anche su piattaforme come Linux, dove la piattaforma Mono

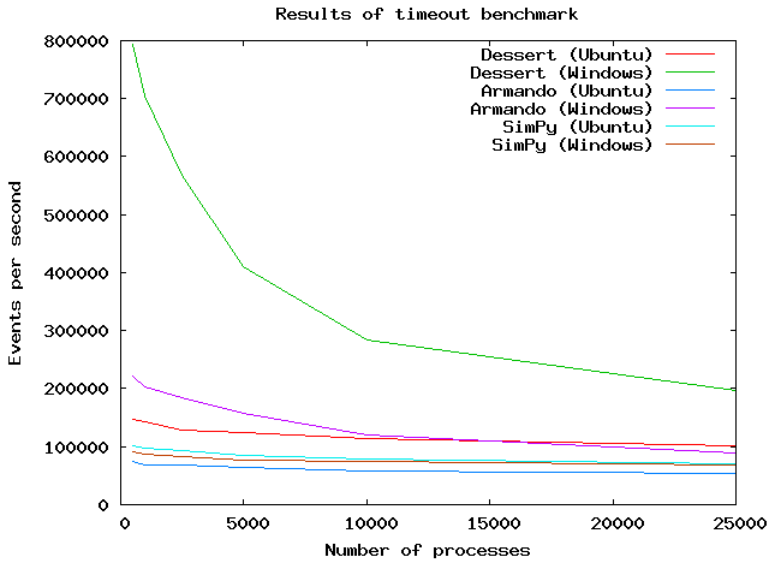


Figura 5.1: Numero di eventi per secondo elaborati dalle varie librerie, all'aumentare dei processi coinvolti.

non è ancora matura e performante come quella .NET. Però, occorre ricordare che quanto abbiamo testato fornisce informazioni solo sul motore, non sulla velocità complessiva nelle simulazioni; infatti, come vedremo, ci saranno delle interessanti sorprese.

Infine, per quanto riguarda il consumo di memoria, mostrato in figura 5.2, SimPy e Dessert si comportano in maniera pressapoco simile, riuscendo a non superare i quaranta megabyte di RAM anche a pieno carico. Dessert su Linux consuma quasi il doppio della memoria, ma questo fatto potrebbe essere dovuto al minor numero di ottimizzazioni offerte dalla piattaforma Mono.

5.2.3 Comportamento dei diversi tipi di heap

Su Dessert abbiamo lasciato aperta la possibilità di specificare il tipo dello heap sottostante il motore, traendo vantaggio dalle diverse scelte offerte da Hippie, la nostra libreria per la gestione degli heap.

Perciò, abbiamo deciso di vedere come si comportassero le varie implementazioni in questo primo confronto; il risultato, piuttosto prevedibile, è mostrato in figura 5.3: lo heap binario, nella sua semplicità implementativa, batta di gran lunga le altre implementazioni, sia su Windows sia su Linux.

Di conseguenza, lo heap binario viene usato di default da Dessert, poiché sembra essere una scelta generalmente ottima.

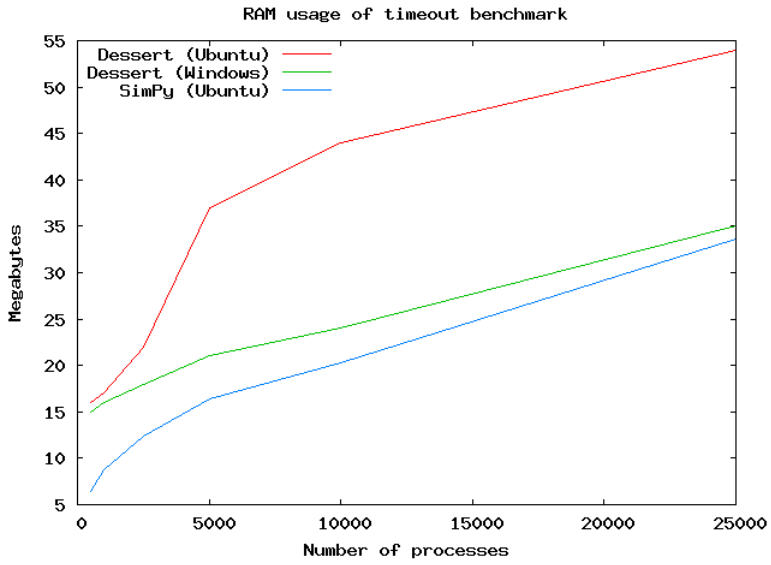


Figura 5.2: Memoria principale usata nel primo confronto, all'aumentare dei processi coinvolti.

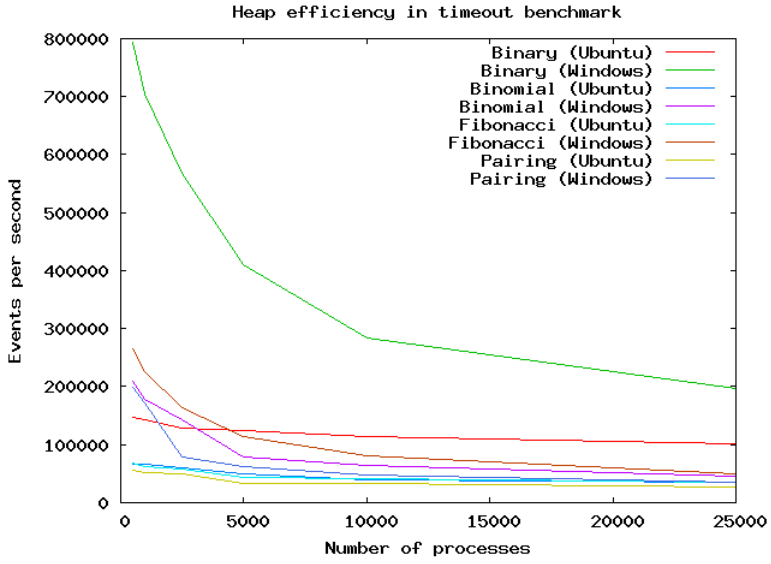


Figura 5.3: Efficienza degli heap nel primo confronto, all'aumentare dei processi coinvolti.

Listato 5.3 Processi produttore e consumatore usati nel secondo test.

```
1 def prodConsBenchmark_Consumer(env, store, counter):
2     while True:
3         yield env.timeout(counter.randomDelay())
4         yield store.get()
5         counter.increment()
6
7 def prodConsBenchmark_Producer(env, store, counter):
8     while True:
9         yield env.timeout(counter.randomDelay())
10        yield store.put(randomInt())
11        counter.increment()
```

5.3 Secondo confronto: produttori e consumatori

Nel primo confronto abbiamo messo alla prova l'efficienza del motore di simulazione, osservando come Dessert e Armando sembrino offrire prestazioni generalmente migliori rispetto a quelle di SimPy. Per la seconda prova abbiamo deciso di riprendere il codice della prima e di aggiungere l'interazione con un oggetto di tipo *store*, registrando il successo degli eventi relativi alle operazioni di inserimento e di rimozione da tale risorsa.

Di fatto, abbiamo ripreso l'esempio del produttore-consumatore e lo abbiamo esteso su larga scala; in particolare, se N è il numero di processi coinvolti nel test, allora $\frac{N}{2}$ sono dedicati al *consumo* della risorsa, mentre i restanti processi sono adibiti alla *produzione* di oggetti da porre nello store. Per chiarezza, nel listato 5.3 mostriamo la definizione di tali processi.

Lo scopo di questo confronto era quello di misurare le prestazioni di parti del simulatore *separate* dal cuore del simulatore stesso, in modo da aggiungere qualche grado di complessità al nostro precedente esempio, il quale era volutamente molto semplice. Pertanto, questa prova ci aiuterà a capire quanto la velocità del motore di simulazione, misurata nel primo confronto, sia indice di velocità *complessiva* del simulatore.

5.3.1 Dati di partenza

I dati di partenza e le modalità di raccolta dei dati finali sono identici a quelli riportati in sezione 5.2.1.

5.3.2 Risultati ottenuti

Come per il primo confronto, occorre spendere due parole sull'andamento delle curve nel grafico in figura 5.4, poiché i risultati continuano a essere degni di discussione. Si può osservare che le curve mantengono una forma pseudo logaritmica, nonostante l'introduzione delle interazioni con un oggetto di tipo *store*;

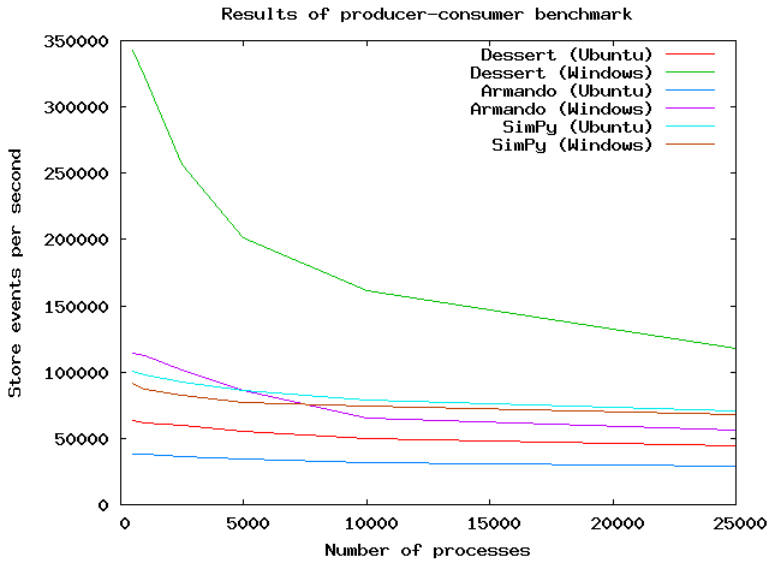


Figura 5.4: Numero di eventi legati allo store, all'aumentare dei processi coinvolti.

perciò, è assolutamente lecito domandarsi perché l'andamento complessivo non cambi.

La spiegazione sta nel fatto che, se ben realizzate, le operazioni sulle code sono eseguibili in $O(1)$, il che non fa salire il grado di complessità della simulazione nel suo complesso. Tuttavia, nonostante la forma delle curve non sia cambiata, è curiosamente mutata la loro posizione reciproca.

Infatti, sebbene Dessert su Windows stia sempre sopra le altre, questa volta notiamo che SimPy sorpassa le prestazioni di Armando su Windows, se i processi sono in gran numero; ciò è una notizia sconfortante, poiché Armando era stato pensato proprio con lo scopo di velocizzare simulazioni esistenti, obiettivo che sembra non essere stato raggiunto. Prima di darci del tutto per vinti, però, attendiamo i risultati del terzo confronto, dove si vedranno le prestazioni *pratiche* dei diversi motori.

Per il resto, si può osservare che Armando su Ubuntu è sempre il fanalino di coda, cosa che può essere causata sia da nostri errori di programmazione, sia da una poco efficiente gestione di IronPython da parte di Mono.

Per concludere l'analisi, abbiamo che Dessert continua a consumare una quantità di memoria principale quasi identica a quella di SimPy, nonostante l'aggiunta di una risorsa condivisa, come mostrato in figura 5.6.

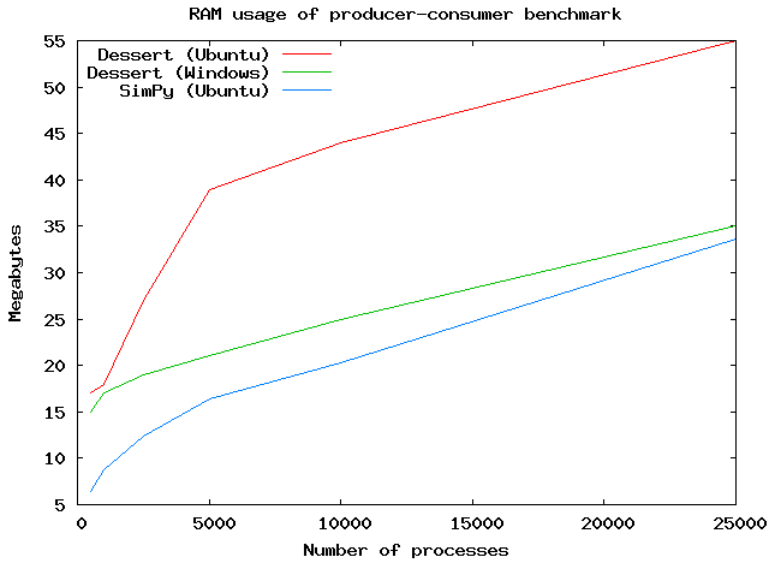


Figura 5.5: Memoria principale usata nel secondo confronto, all'aumentare dei processi coinvolti.

5.3.3 Comportamento dei diversi tipi di heap

Anche nei dati prodotti dall'esempio del produttore consumatore, riportati per chiarezza in figura 5.6, notiamo che lo heap binario sovrasta le altre implementazioni, sia su Windows sia su Ubuntu.

Pertanto, abbiamo deciso di non eseguire un test simile per il terzo confronto, dato che i risultati sarebbero stati facilmente immaginabili; lo heap usato per la simulazione del sistema peer to peer sarà di tipo binario, con il quale Dessert offre le prestazioni migliori.

5.4 Terzo confronto: codifica dei pacchetti

Per il terzo e ultimo confronto, abbiamo ripreso una simulazione che avevamo scritto come progetto per il corso di *Social and Peer to Peer Networks* e la abbiamo riscritta in modo che funzionasse sia su Dessert, sia su SimPy 3. Quella simulazione, riguardante l'applicazione dei campi di Galois alla codifica dei pacchetti di rete, era stata progettata per la versione 2.3 di SimPy; considerato che la versione 3 non è compatibile con la precedente, non abbiamo potuto riusare immediatamente quel codice, ma lo abbiamo dovuto sistemare, in modo che fosse allineato con le nuove specifiche.

La simulazione consiste nell'analisi del comportamento di un sistema *peer to peer*, composto da un certo numero di nodi che agiscono sia come client, sia come server; i client memorizzano dei file, o meglio, dei pacchetti, sui server, i

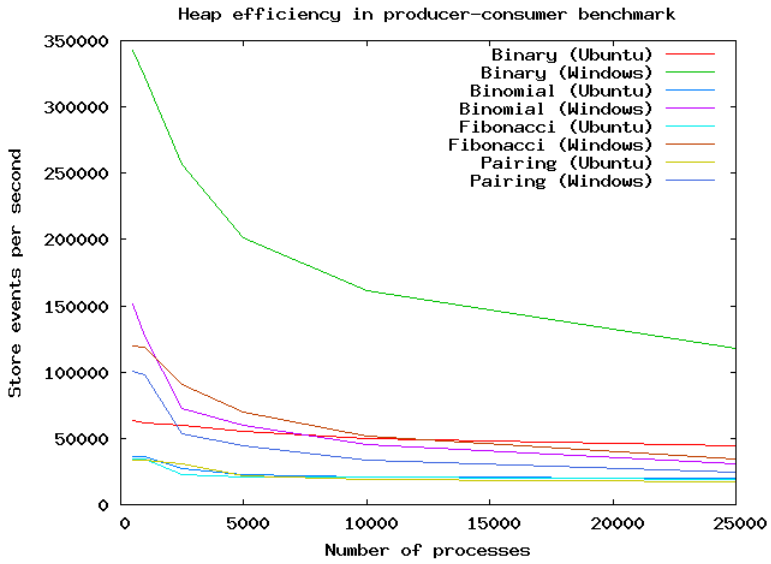


Figura 5.6: Efficienza degli heap nel secondo confronto, all'aumentare dei processi coinvolti.

quali dovranno restare in attesa che i client chiedano nuovamente quanto hanno caricato.

Inizialmente, si parte con un sistema dove i client hanno già memorizzato i loro file sui server. In particolare, i file, prima di essere caricati, vengono divisi in n pacchetti più piccoli; tali pacchetti vengono, a loro volta, codificati secondo un codice a cancellazione [133], con il quale si otterranno $2n-1$ nuovi pacchetti. Essi vengono spediti ai server, in modo che ciascun server riceva un pacchetto differente.

Dato che i pacchetti sono stati prodotti da un codice a cancellazione, per ricostruire il file sarà sufficiente recuperare n pacchetti codificati, in un ordine qualunque, purché siano distinti.

La simulazione, che avviene dopo la memorizzazione dei file, consiste nel recupero dei pacchetti da parte dei client, nell'ipotesi che la rete sia piuttosto congestionata e che sia facile che molti pacchetti vadano perduti. La congestione della rete è causata dal fatto che tutte le connessioni passano da un unico switch, il quale può memorizzare un numero limitato di frame nel proprio buffer; perciò, se le richieste (e le relative risposte) sono in un numero eccessivo, lo switch dovrà rigettare molti frame, poiché non vi sarà posto sufficiente nel buffer.

Quindi, mettendoci nei panni del client, ci troviamo nella situazione di dover recuperare almeno n pacchetti, ma di non poter effettuare troppe richieste, poiché, se così facessimo, la rete diverrebbe intasata e rischieremmo di dover impiegare molto tempo prima di riuscire a recuperare il file memorizzato. La simulazione si occupa esattamente di capire quante richieste causino una situa-

zione ottimale, dove i client possono recuperare i pacchetti nel minor tempo possibile e dove il numero complessivo di pacchetti recuperati è il maggiore.

5.4.1 Processi coinvolti

La simulazione appena descritta consta di cinque processi differenti:

- Lo switch, attraverso il quale passano tutti i pacchetti UDP. Esso, avendo un buffer limitato, sarà la causa della perdita di molti pacchetti in transito sulla rete.
- I client, il cui scopo è quello di recuperare i file memorizzati. Essi si occuperanno di inoltrare le richieste ai server e di controllare che i file siano effettivamente recuperati con successo.
- La parte a basso livello dei client, che si prenderà carico di gestire l'invio dei pacchetti UDP, contenenti le richieste per i server, e si occuperà della ricostruzione dei file.
- I server, i quali processeranno le richieste giunte dai client.
- La parte a basso livello dei server, che gestirà la conversione dei pacchetti in entrata e in uscita verso il server.

Vi sono molti altri dettagli relativi al comportamento di quei processi, ma la loro descrizione esula dagli scopi di questo capitolo. Infatti, vorremmo soltanto che fosse chiaro che questa è una simulazione mediamente complessa, in quanto si fa uso di diversi costrutti definiti dal paradigma di SimPy, e che aveva molto senso usarla come riferimento per la valutazione delle prestazioni del nostro lavoro.

Detto ciò, non discuteremo dettagliatamente dei risultati che tale simulazione produce, ma ci limiteremo soltanto a commentare i grafici riportati in figura 5.7 e 5.8. Lì possiamo osservare i dati ottenuti da una simulazione con 32 macchine, relativi rispettivamente al tempo medio di attesa dei client per il recupero di un file e al numero complessivo di file ricostruiti con successo nel corso di una simulazione.

Nei grafici si nota immediatamente che abbiamo i minimi tempi di attesa e la massima efficienza nel ricostruire i file quando le richieste aggiuntive sono tre o quattro. Pertanto, la simulazione dimostra la propria utilità suggerendoci un'idea leggermente contro intuitiva, ovvero, che l'inviare un piccolo numero di richieste superiori al dovuto migliora il rendimento del sistema. Anche se tale approccio comporterà una maggiore perdita di pacchetti da parte dello switch, come mostrato in figura 5.9, complessivamente il sistema diverrà più efficiente e i client potranno recuperare i file in minor tempo.

5.4.2 Dati di partenza

La simulazione coinvolge un numero prefissato di macchine, compreso nell'insieme formato da $\{4, 8, 16, 32\}$; ad ogni quantità di macchine è associato un certo

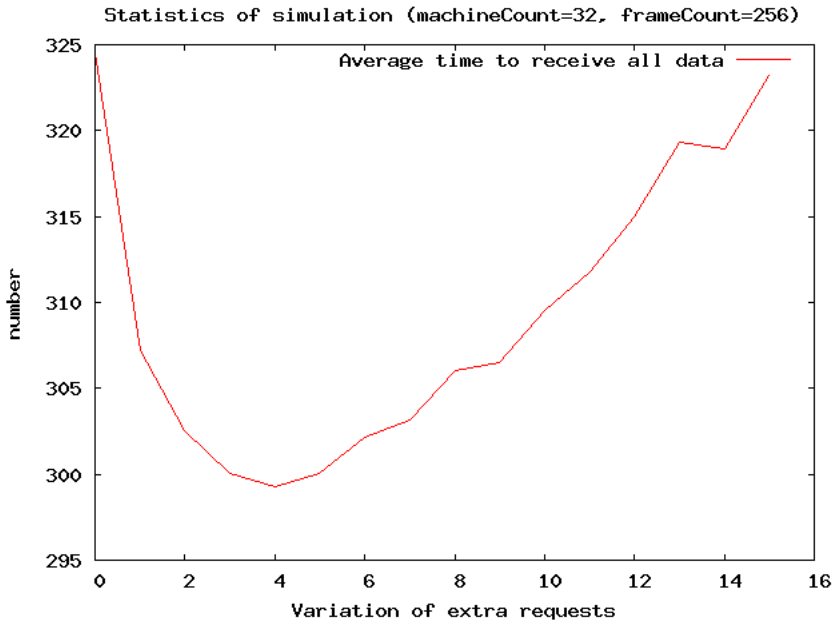


Figura 5.7: Tempo medio di attesa dei client, al variare delle richieste aggiuntive.

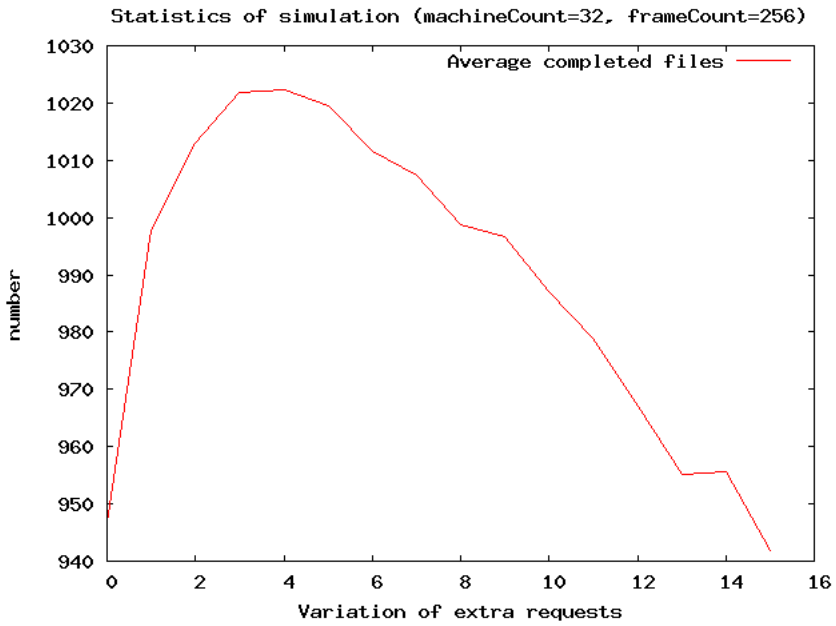


Figura 5.8: Numero medio di file ottenuti globalmente dai client nel corso della simulazione, al variare delle richieste aggiuntive.

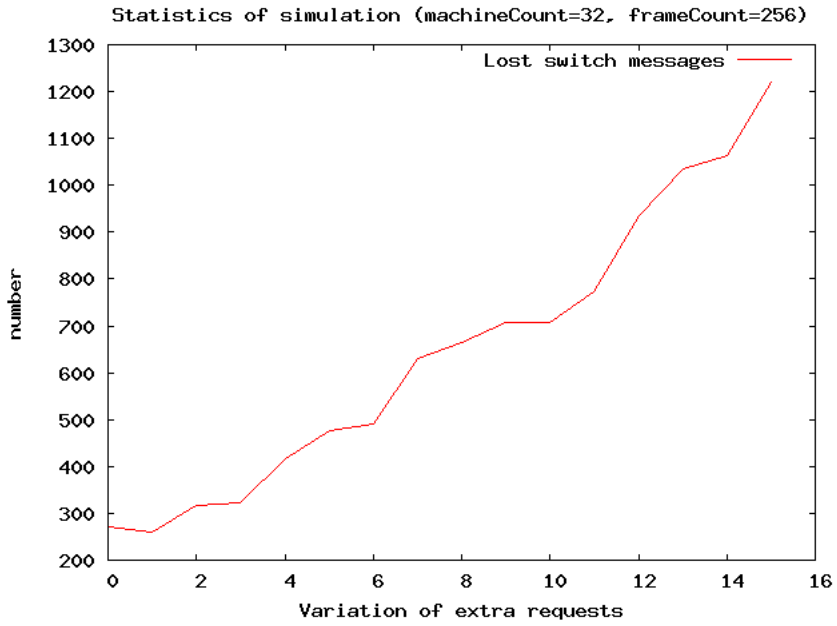


Figura 5.9: Numero medio di pacchetti persi dallo switch, al variare delle richieste aggiuntive.

numero di frame per lo switch, in modo che non vi siano troppi pacchetti persi già nelle fasi iniziali della simulazione.

Preso un certo numero di macchine M, vogliamo analizzare il comportamento della rete aggiungendo da zero a $\frac{M}{2} - 1$ richieste alle $\frac{M}{2}$ che i client dovranno comunque fare. Considerato che, per ogni configurazione, vengono eseguite venti simulazioni al fine di ottenere dei dati affidabili, abbiamo che il numero totale di simulazioni eseguite è, per ogni M, $10 \cdot M$.

In merito al numero totale di processi effettivamente presenti, occorre ricordare che ogni macchina ne è descritta da quattro (il client, il server e i relativi "sistemi operativi") e che il processo dello switch è sempre presente. Quindi, il nostro test parte da 17 processi, nel caso più semplice, per arrivare a 129, nel caso con più macchine.

Le venti simulazioni da eseguire per ogni quantità di richieste aggiuntive vengono ripartite su quattro thread, al fine di ridurre sensibilmente i tempi di esecuzione; nel caso di Python, però, mostriamo anche come la loro presenza rallenti, sorprendentemente, la velocità di esecuzione del test. Ciò è dovuto al *global interpreter lock* [16], il quale impedisce una vera e propria programmazione multithreaded sull'interprete standard, *CPython*.

5.4.3 Risultati ottenuti

I risultati, mostrati in figura 5.10, sono la conferma che il nostro lavoro ha, fortunatamente, raggiunto i propri obiettivi. Dalla lettura del grafico è possibile notare come Dessert, su entrambe le piattaforme, sia la libreria più veloce, riuscendo a completare, nel caso migliore, le simulazioni per le 32 macchine in circa 30 minuti.

A seguire troviamo Armando su Windows, un'altra nota positiva per il nostro progetto; dopodiché, troviamo SimPy su Ubuntu, impostata in maniera tale da non usare thread. Il layer di traduzione sotto Ubuntu conferma i propri difetti, piazzandosi tra gli ultimi anche in questo test.

Risulta incredibile osservare come l'uso dei thread su Python diminuisca pesantemente le prestazioni, in particolare se si usa Windows: il ciclo di simulazioni con 32 macchine, su tale sistema operativo, tocca le cinque ore di durata, cosa che impedisce assolutamente di scalare la simulazione tramite l'aumento del numero di macchine attive.

Su questo esempio, che rappresentava una reale simulazione di media scala, abbiamo deciso di vedere quale impatto avesse la rimozione dei controlli di integrità, cosa che è fattibile grazie all'uso della nostra libreria Thrower; in particolare, abbiamo eseguito il test solo su Windows, confrontando Dessert e Armando con e senza l'uso dei controlli di integrità. I risultati, visibili in figura 5.11, dimostrano che vi è soltanto un flebile incremento delle prestazioni, che diventa più cospicuo, ma non decisamente rilevante, al crescere della dimensione della simulazione.

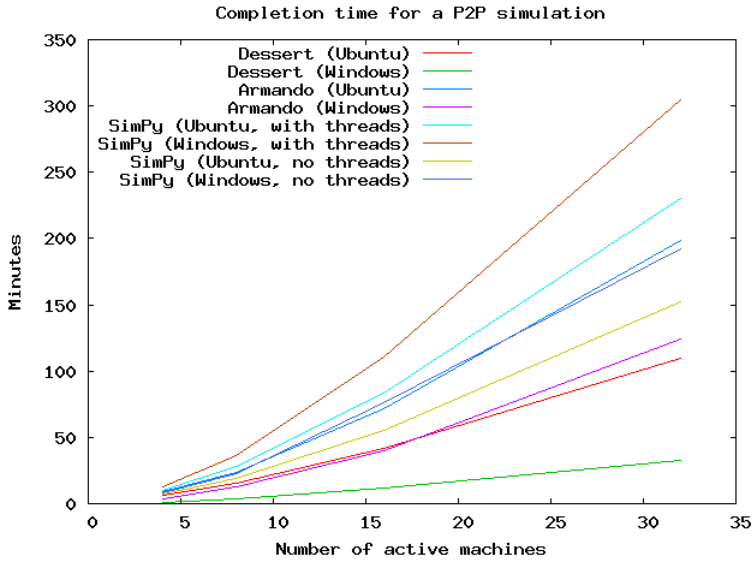


Figura 5.10: Tempo impiegato dalle varie simulazioni sulle piattaforme prese in esame.

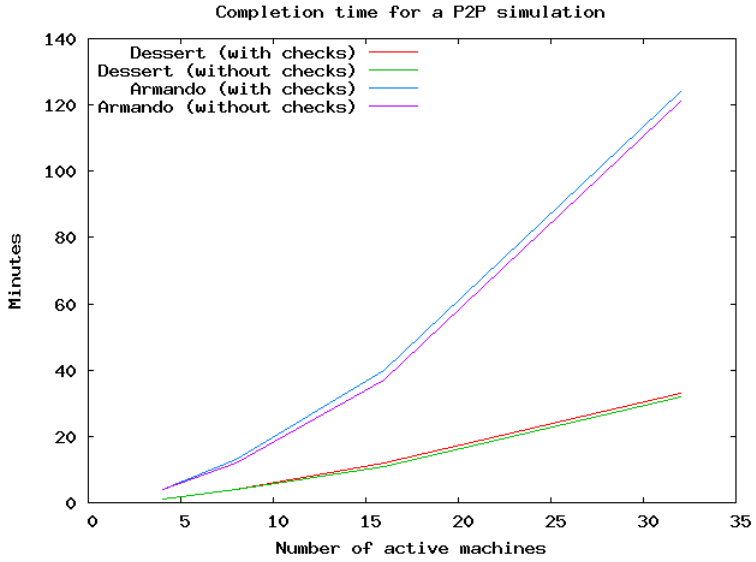


Figura 5.11: Test di Dessert e Armando con e senza controlli di integrità.

Capitolo 6

Conclusioni

Chiudiamo questa relazione con una veloce carrellata di ciò che è stato fatto, di ciò che *non* è stato fatto e di ciò che, in futuro, sarebbe interessante fare.

Lo sviluppo di Dessert è stato lungo e tortuoso, e molte parti di esso non hanno trovato posto, per questioni di sinteticità, in questa relazione; perciò, approfitteremo di questo capitolo finale per accennare a *tutte* le parti del progetto, di modo che il lettore possa avere una visione più completa di ciò che è stato fatto. Tuttavia, indipendentemente dalle difficoltà e dalle problematiche incontrate, occorre dire che la costruzione del simulatore, e delle librerie a esso collegate, è stata indubbiamente un'interessante esperienza, che ci ha permesso di scavare a fondo nelle possibilità presenti nel *mondo* .NET.

In sezione 6.1 vedremo rapidamente cosa sia stato effettivamente fatto; dopodiché, in sezione 6.2, faremo alcuni cenni alle parti che non siamo riusciti a completare e, infine, in sezione 6.3, concluderemo discutendo alcune estensioni future.

6.1 Lavoro svolto

Scopo del nostro progetto è stato quello di creare un *clone* su .NET della libreria SimPy, al fine di ottenere un simulatore decisamente più efficiente, senza sacrificare nulla in termini di design. I risultati, esposti nel capitolo 5, hanno confermato che, almeno sui sistemi Windows, gli obiettivi sono stati ampiamente raggiunti; su Linux, invece, abbiamo ottenuto dei buoni risultati, ma potenzialmente vi è ancora parecchio margine di miglioramento.

Il lavoro su Dessert ha fatto nascere tutta una serie di progetti *satellite*, molti dei quali descritti nel capitolo 3; tali progetti di utilità generale sono indipendenti da Dessert, che ne è un *client*, e sono stati pubblicati in rete per l'uso da parte di terzi. Molti di quei progetti hanno raggiunto un discreto livello di maturità e di affidabilità, a causa della necessità di avere del codice altamente sicuro ed efficiente da poter integrare all'interno del simulatore.

La scelta di creare Armando, un layer di traduzione verso le simulazioni scritte in Python, ci ha "obbligati" a studiare a fondo le funzionalità offerte da IronPython, cosa che ha giovato molto alla nostra esperienza d'uso nei confronti di tale libreria. Infatti, abbiamo compreso a fondo le interazioni tra l'interprete, il codice Python e il codice .NET, ottenendo le conoscenze necessarie per esporre la nostra libreria, completamente scritta in C#, come se fosse una libreria "nativa" per Python.

Realizzare alcune caratteristiche, come la corretta gestione dei generatori e delle eccezioni, ha richiesto una notevole quantità di studio e sperimentazione; in ogni caso, i risultati ottenuti hanno ripagato gli sforzi fatti. Per testare la correttezza del nostro lavoro abbiamo anche cercato di utilizzare, tramite un apposito *test runner* scritto da noi, direttamente gli unit test di SimPy, anche se ciò non ha avuto molto successo per via di diverse questioni tecniche. Ad ogni modo, il lavoro che abbiamo iniziato potrà essere ripreso in futuro e, opportunamente completato, offrirà un sistema molto pratico per controllare la qualità del layer.

Infine, per questa relazione, ma non solo, abbiamo scritto un buon numero di esempi in vari linguaggi per la piattaforma .NET, anche usandone alcuni innegabilmente "esotici", come Boo e F#. Questi, oltre ad aiutare il rapido apprendimento della libreria, sono un punto di riferimento importante per accertarsi che il nostro lavoro rispetti i propri obiettivi e non si discosti eccessivamente dal nostro modello, la libreria SimPy. Oltre agli esempi, è presente una batteria di unit test, il cui scopo è sia quello di mantenere la correttezza della nostra implementazione, sia quello di contribuire alla verifica continua della pulizia del design del progetto.

6.2 Parti da completare

Dessert, per quanto funzionante, non ha il grado di maturità e completezza per essere considerata un prodotto finito e richiederebbe ulteriore lavoro per curarne i dettagli e la relativa documentazione. Inoltre, sarebbe utile sviluppare ulteriori test, idealmente condivisi con la libreria SimPy, in modo tale da assicurare l'eguale funzionamento delle due librerie.

Considerato che Dessert, per come è stata progettata, avrebbe il potenziale per essere utilizzata a livello internazionale, un'opportuna ed estesa documentazione in inglese, tutt'ora mancante, le faciliterebbe il raggiungimento di tale obiettivo. Nel momento in cui questa relazione è stata scritta, la libreria non è né munita di una documentazione allegata al codice, né di guide per l'uso o di sintetiche istruzioni per agevolare il passaggio a Dessert da parte di chi già conosce SimPy.

Un altro obiettivo da perseguire in lavori futuri dovrebbe essere quello di continuare a garantire la parità e la compatibilità nei confronti della libreria SimPy. Ad esempio, allo stato attuale, Armando richiederebbe ulteriore lavoro per assicurare la maggior copertura possibile verso le simulazioni scritte in Py-

Listato 6.1 Esempio di un possibile DSL per Dessert.

```
process car(env) =
    while true do
        print("Start parking at {0}", env.Now)
        parkingDuration = 5
        wait env.Timeout(parking_duration)
        print("Start driving at {0}", env.Now)
        tripDuration = 2
        wait env.Timeout(tripDuration)

env = Environment()
env.Start(car(env))
env.Run(until=15)
```

thon; in particolare, sarebbe utile migliorare la segnalazione degli errori, che è piuttosto carente.

Infine, una parte che abbiamo completamente tralasciato, più per mancanza di interesse che per questioni tecniche, è la gestione delle simulazioni in tempo reale, esposta dalla libreria SimPy nel modulo `simpy.rt`.

6.3 Possibili estensioni

Fra le possibili estensioni del nostro lavoro citiamo l'introduzione di un *domain specific language* (DSL) e la preparazione di ulteriori librerie, contenenti processi e componenti ad alto livello da usare nelle nuove simulazioni. Idealmente, le due *estensioni* potrebbero collaborare l'una al successo dell'altra: ad esempio, il DSL potrebbe facilitare l'uso e/o la scrittura delle nuove componenti ad alto livello.

Un DSL potrebbe offrire una sintassi snella e semplificata, nella quale molti *fronzoli* presenti in Dessert, come lo statement `yield return`, vengano gestiti in automatico dal DSL stesso. Nel listato 6.1 presentiamo un piccolo esempio di ciò che intenderemmo ottenere con un linguaggio di quel tipo; si noti come molte parti, tra cui la dichiarazione del tipo di ritorno e l'uso di `yield`, siano sparite dal codice, in quanto esse sono maggiormente legate ad aspetti implementativi che non alla dichiarazione della simulazione stessa. Il codice mostrato nell'esempio è basato su quello riportato nei listati 2.1 e 4.1.

Per quanto riguarda la creazione di componenti ad alto livello, invece, ci riferiamo alla necessità di costruire dei processi che simulino oggetti di comune interesse in vari ambiti, come, ad esempio:

- I router, gli switch, le schede di rete e i protocolli di rete più usati, per le simulazioni inerenti le reti.
- Dei processi che simulino il funzionamento di magazzini, centri di smistamento, etc etc, per il settore manifatturiero.

- Componenti relative ai mezzi pubblici, al traffico stradale e all'affollamento pedonale, per l'ambito civile.

Quegli "oggetti", scritti come processi Dessert, eventualmente sfruttando una sintassi semplice offerta dal DSL, potrebbero essere facilmente integrati in nuove simulazioni ancora più complesse, le quali non si dovrebbero fare carico della manutenzione e della scrittura delle parti comuni e famose, già pronte come estensioni.

Oltretutto, una funzionalità simile aprirebbe la strada a interfacce grafiche ad altissimo livello, le quali potrebbero esporre dei paradigmi grafici per realizzare simulazioni rapidamente e senza doversi preoccupare dei diversi dettagli implementativi.

Capitolo 7

Ringraziamenti

Scrivere i ringraziamenti pare sia una cosa ancora più complicata del resto della tesi, poiché si corre sempre il rischio di dimenticarsi di qualcuno. Tuttavia, nel caso un "qualcuno" si senta offeso, non avendo trovato il suo nome nei ringraziamenti e aspettandosi che ci fosse, lo prego di scusarmi: ho scritto questa parte alla fine del lavoro e la mia mente era in un serio stato di precarietà!

Quindi, ora ci saranno i vari ringraziamenti a tutti coloro che hanno contribuito, più o meno direttamente, alla realizzazione di questo progetto e a tutti quelli che mi hanno accompagnato nella lunga avventura universitaria. Mettetevi comodi, la lista è lunga e non segue un ordine particolare:

- I professori Giovanni Lagorio, Maura Cerioli, Davide Ancona, Marina Ribaudo e Giovanni Chiola, per aver tenuto i corsi che ho seguito con maggiore passione e interesse; gli argomenti da loro trattati sono stati tra i pochi che ho deciso di approfondire per conto mio, oltre quanto fosse richiesto per il superamento dell'esame. Tra loro, però, un ringraziamento ancora più sentito va a Giovanni Lagorio, che mi ha seguito nel lungo percorso che ha portato a questa tesi. Egli mi ha assecondato nelle mie strambe avventure nel mondo .NET, ponendosi più come un compagno di studi che non come un professore e insegnante. Se non ricordo male, questa è la prima tesi specialistica che ha seguito; nel caso, spero che non ne sia rimasto sconvolto e che continui a seguire molti altri studenti, poiché di sicuro ne troverà di più intelligenti! Anche se dubito molto fortemente che ne troverà di più pazzi (vedi Armando, Dessert, Hippie e altri nomi furbi e significativi!)...
- Il professor Eugenio Moggi, per avere tenuto il corso più complicato che io abbia mai visto e immaginato. Direi che è stato il primo e l'ultimo corso che ho dovuto letteralmente abbandonare, quindi un inchino a lei e a chiunque sia riuscito a passarlo! In ogni caso, il teorema di Rice lo avevo capito bene per davvero, ma sfortunatamente è tra le poche cose che ricordo...

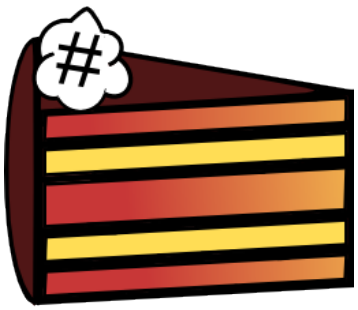


Figura 7.1: Il logo di Dessert creato da Ilaria, mia cuginetta.

- I compagni di corso della triennale, Marta, Federica, Chiara, Artur, Laura, Andrea (Bracco), Niccolò (Icco), Mirko, Luca, Lorenzo. Con loro i tre anni sono volati, perché hanno contribuito alla creazione di un piacevolissimo ambiente di studio. Non abbiamo seguito molte lezioni perché avevamo da parlare e da giocare con pupazzetti e calamite, ma è stato proprio quello a rendere tutto più bello. Poi, come avremmo potuto seguire con i mitici disegni di Mirko proprio sotto il naso, mentre Icco raccontava le mirabolanti avventure dei suoi improbabili amici? In questo caso, ci sono due ringraziamenti speciali da fare. Uno per Artur, con il quale all'interno del dipartimento ho svolto ogni tipo di attività (lecita, come progetti e ripasso, e illecita, come apertura di frigoriferi e altro!), e uno per Bracco, che ha sempre cercato di tenerci tutti uniti, anche dopo la fine della triennale, organizzando insistentemente aperitivi ogni due o tre settimane.

- I compagni di corso della magistrale, Artur, Paolo Farina, Paolo Macco, Mattia; non aggiungo Francesco (F) perché non se lo merita! Con loro, oltre alle consuete difficoltà dello studio, abbiamo anche affrontato diversi pranzi in mensa, resistendo al cibo un po' immangiabile, alle pessime battute di Paolo Macco e Francesco e sentendo gli strani e perversi discorsi di Paolo Farina. Tutto questo, mentre Mattia desiderava ardentemente tutte le spinacine al di là del bancone!

- Ilaria, mia cuginetta, per aver creato il bellissimo logo per Dessert che abbiamo riportato in figura 7.1. Hai fatto decisamente presto a imparare a usare InkScape, speriamo che tu possa continuare a perseguire con successo la tua passione "artistica"!

- Guido, l'unico amico del liceo che sia rimasto tale anche tanti anni dopo. Nonostante la mia asocialità, gli impegni e tutto il resto, ha sempre cercato di farsi sentire e di organizzare "gustose" passeggiate in giro per Chiavari, e non solo.

- Radio Deejay, in particolare il programma "Chiamate Roma Triuno Triuno", per avermi fatto compagnia durante le altrimenti silenziose giornate passate alla casa dello studente.

- Celestino, Fragolino e Johnny, i computer con i quali ho affrontato le sfide poste durante questi cinque lunghi anni. Tutti e tre hanno dovuto subire le angherie di Visual Studio, terminali e compilatori vari, senza battere minimamente ciglio! Quando sono stati prodotti, saranno stati convinti di finire a riprodurre musica o altro... Invece, siete finiti a fare tante cose brutte!

Appendice A

Esempi della banca

I listati A.1 e A.2 riportano per esteso gli esempi della banca mostrati rispettivamente nelle sezioni 2.6 e 4.5, in modo che il lettore possa avere una visione di insieme di quanto viene raccontato in tali sezioni.

Inoltre, ricordiamo che, nel caso si vogliano provare tali esempi per conto proprio, i loro codice è presente anche sul repository del progetto [112].

Listato A.1 Esempio della banca, implementato in Python su SimPy.

```
rand = Random(21)
exp = lambda x: rand.expovariate(1.0/x)

avgIncomingT, avgServiceT = 3.0, 10.0 # Minuti
queueCount = 3 # Numero sportelli
bankCap, bankLvl = 20000, 2000 # Euro

totClients, totServedClients = 0, 0
totWaitT, totServT = 0.0, 0.0

def client(env, queue, bank, amount, get):
    global totServedClients, totWaitT, totServT
    with queue.request() as req:
        start = env.now
        yield req
        totWaitT += env.now - start
        start = env.now
        yield env.timeout(exp(avgServiceT))
        if get: yield bank.get(amount)
        else: yield bank.put(amount)
        totServT += env.now - start
        totServedClients += 1

def spawner(env, queues, bank):
    global totClients
    while True:
        yield env.timeout(exp(avgIncomingT))
        queue = min(queues, key=lambda q:len(q.queue))
        amount = rand.uniform(50, 500)
        get = rand.random() < 0.4
        env.start(client(env, queue, bank, amount, get))
        totClients += 1

env = Environment()
queues = [Resource(env, 1) for i in range(queueCount)]
bank = Container(env, bankCap, bankLvl)

env.start(spawner(env, queues, bank))
env.run(until=5*60)

lvl = bank.level
print("Finanze totali al tempo %.2f: %d" % (env.now, lvl))
print("Clienti entrati: %d" % totClients)
print("Clienti serviti: %d" % totServedClients)
avgWait = totWaitT/totServedClients
print("Tempo medio di attesa: %.2f" % avgWait)
avgServ = totServT/totServedClients
print("Tempo medio di servizio: %.2f" % avgServ)
```

Listato A.2 Esempio della banca, implementato in F# su Dessert.

```
Sim.CurrentTimeUnit <- TimeUnit.Minute
let avgIncomingT, avgServiceT = (3).Minutes(), (10).Minutes()
let queueCount = 3 // Numero sportelli
let bankCap, bankLvl = 20000.0, 2000.0 // Euro
let waitTally, servTally = Sim.NewTally(), Sim.NewTally()
let mutable totClients = 0

let client(env: IEnvironment, queue: IResource,
           bank: IContainer, amount, get) = seq<IEvent> {
    use req = queue.Request()
    let s1 = env.Now
    yield upcast req
    waitTally.Observe(env.Now - s1)
    let s2 = env.Now
    yield upcast env.Timeout(env.Random.Exponential(1.0/avgServiceT))
    if get then yield upcast bank.Get(amount)
    else yield upcast bank.Put(amount)
    servTally.Observe(env.Now - s2)
}

let rec spawner(env: IEnvironment, queues: IResource list,
                bank) = seq<IEvent> {
    yield upcast env.Timeout(env.Random.Exponential(1.0/avgIncomingT))
    let queue = queues.MinBy(fun q -> q.Count)
    let amount = float(env.Random.Next(50, 500))
    let get = env.Random.NextDouble() < 0.4
    env.Start(client(env, queue, bank, amount, get)) |> ignore
    totClients <- totClients + 1
    yield! spawner(env, queues, bank)
}

let run() =
    let env = Sim.NewEnvironment(seed = 21)
    let queues = [for x in 1 .. queueCount do
                      yield env.NewResource(1)]
    let bank = env.NewContainer(bankCap, bankLvl)

    // Avvio della simulazione
    env.Start(spawner(env, queues, bank)) |> ignore
    env.Run(until = (5).Hours())

    // Raccolta dati statistici
    let lvl = bank.Level
    printfn "Finanze totali al tempo %.2f: %g" env.Now lvl
    printfn "Clienti entrati: %d" totClients
    printfn "Clienti serviti: %d" servTally.Count
    printfn "Tempo medio di attesa: %.2f" (waitTally.Mean())
    printfn "Tempo medio di servizio: %.2f" (servTally.Mean())
```

Appendice B

Esempi per Dessert

In questa appendice riportiamo le traduzioni su Dessert di molti esempi per SimPy, i quali sono presenti nel capitolo 2. Le traduzioni sono state poste qui, piuttosto che nel capitolo dedicato a Dessert, per non appesantirne troppo la lettura.

Gli esempi non hanno bisogno di particolari commenti, poiché sono molto simili a quelli per SimPy, escluse le inevitabili differenze sintattiche.

Listato B.1 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.7.

```
1  static class PastaCooking {
2      static readonly double AvgCookTime;
3      static readonly double StdCookTime;
4      static readonly double SimTime;
5
6      static PastaCooking() {
7          Sim.CurrentTimeUnit = TimeUnit.Minute;
8          AvgCookTime = 10.Minutes();
9          StdCookTime = 1.Minutes();
10         SimTime = 50.Minutes();
11     }
12
13     static IEnumerable<IEvent> PastaCook(IEnvironment env) {
14         while (true) {
15             var cookTime = env.Random.Normal(AvgCookTime,
16                                              StdCookTime);
17             var fmt = "Pasta in cottura per {0} minuti";
18             Console.WriteLine(fmt, cookTime);
19             yield return env.Timeout(cookTime);
20             if (cookTime < AvgCookTime - StdCookTime)
21                 Console.WriteLine("Pasta poco cotta!");
22             else if (cookTime > AvgCookTime + StdCookTime)
23                 Console.WriteLine("Pasta troppo cotta...");
24             else
25                 Console.WriteLine("Pasta ben cotta!!!");
26         }
27     }
28
29     public static void Run() {
30         var env = Sim.NewEnvironment(21);
31         env.Start(PastaCook(env));
32         env.Run(SimTime);
33     }
34 }
```

Listato B.2 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.9.

```
static class TargetShooting {
    const double HitProb = 0.7;
    const double SimTime = 100;

    static string[] Targets = {"Alieno", "Pollo", "Unicorno"};

    static ITimeout<string> NewTarget(IEnvironment env) {
        var delay = env.Random.DiscreteUniform(1, 20);
        var target = env.Random.Choice(Targets);
        return env.Timeout(delay, target);
    }

    static IEnumerable<IEvent> Shooter(IEnvironment env) {
        while (true) {
            var timeout = NewTarget(env);
            yield return timeout;
            var hit = env.Random.NextDouble();
            string fmt;
            if (hit < HitProb)
                fmt = "{0}: {1} colpito, si!";
            else
                fmt = "{0}: {1} mancato, no...";
            Console.WriteLine(fmt, env.Now, timeout.Value);
        }
    }

    public static void Run() {
        var env = Sim.NewEnvironment(21);
        env.Start(Shooter(env));
        env.Run(SimTime);
    }
}
```

Listato B.3 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.11.

```
static class MachineLoad {
    static readonly string[] Tasks = {"A", "B", "C"};
    static readonly double LoadTime;
    static readonly double WorkTime;
    static readonly double SimTime;

    static MachineLoad() {
        Sim.CurrentTimeUnit = TimeUnit.Minute;
        LoadTime = 5.Minutes();
        WorkTime = 25.Minutes();
        SimTime = 100.Minutes();
    }

    static IEnumerable<IEvent> Worker(IEnvironment env) {
        Console.WriteLine("{0}: Carico la macchina...",
                          env.Now);
        yield return env.Timeout(LoadTime);
        env.Exit(env.Random.Choice(Tasks));
    }

    static IEnumerable<IEvent> Machine(IEnvironment env) {
        while (true) {
            var worker = env.Start(Worker(env));
            yield return worker;
            Console.WriteLine("{0}: Eseguo il comando {1}",
                              env.Now, worker.Value);
            yield return env.Timeout(WorkTime);
        }
    }

    public static void Run() {
        var env = Sim.NewEnvironment(21);
        env.Start(Machine(env));
        env.Run(SimTime);
    }
}
```

Listato B.4 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.13.

```
1  static class EventTrigger {
2      static IEnumerable<IEvent> DoSucceed(IEnvironment env,
3                                           IEvent<object> ev)
4      {
5          yield return env.Timeout(5);
6          ev.Succeed("SI :)");
7      }
8
9      static IEnumerable<IEvent> DoFail(IEnvironment env,
10                                       IEvent<object> ev)
11     {
12         yield return env.Timeout(5);
13         ev.Fail("NO :(");
14     }
15
16     static IEnumerable<IEvent> Proc(IEnvironment env) {
17         var ev1 = env.Event();
18         env.Start(DoSucceed(env, ev1));
19         yield return ev1;
20         if (ev1.Succeeded)
21             Console.WriteLine(ev1.Value);
22
23         var ev2 = env.Event();
24         env.Start(DoFail(env, ev2));
25         yield return ev2;
26         if (ev2.Failed)
27             Console.WriteLine(ev2.Value);
28     }
29
30     public static void Run() {
31         var env = Sim.NewEnvironment();
32         env.Start(Proc(env));
33         env.Run();
34     }
35 }
```

Listato B.5 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.14.

```
Module EventCallbacks
    Iterator Function DoFail(env As IEnvironment,
                             ev As IEvent(Of String))
        As IEnumerable(Of IEvent)
        Yield env.Timeout(5)
        ev.Fail("NO")
    End Function

    Sub MyCallback(ev As IEvent)
        Console.WriteLine("Successo: '{0}'; Valore: '{1}'",
                          ev.Succeeded, ev.Value)
    End Sub

    Iterator Function Proc(env As IEnvironment)
        As IEnumerable(Of IEvent)
        Dim ev1 = env.Timeout(7, "SI")
        ev1.Callbacks.Add(AddressOf MyCallback)
        Yield ev1
        Dim ev2 = env.Event(Of String)()
        ev2.Callbacks.Add(AddressOf MyCallback)
        env.Start(DoFail(env, ev2))
        Yield ev2
    End Function

    Sub Run()
        Dim env = Sim.NewEnvironment()
        env.Start(Proc(env))
        env.Run()
    End Sub
End Module
```

Listato B.6 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.15.

```
static class ConditionTester {
    static IEnumerable<IEvent> AProcess(IEnvironment env) {
        yield return env.Timeout(7);
        env.Exit("VAL_P");
    }

    static IEnumerable<IEvent> CondTester(IEnvironment env) {
        var t1 = env.NamedTimeout(5, "VAL_T", "T");
        var aProc = env.Start(AProcess(env), name: "P");
        var cond = env.AllOf(t1, aProc);
        yield return cond;
        Console.WriteLine("ALL: {0}", cond.Value);

        var t2 = env.NamedTimeout(5, "VAL_T", "T");
        aProc = env.Start(AProcess(env), name: "P");
        cond = env.AnyOf(t2, aProc);
        yield return cond;
        Console.WriteLine("ANY: {0}", cond.Value);

        aProc = env.Start(AProcess(env), name: "P");
        var aTime = env.NamedTimeout(5, "VAL_T", "T");
        ConditionEval<ITimeout<string>, IProcess> pred =
            c => c.Ev1.Succeeded && c.Ev2.Succeeded &&
            c.Ev1.Value.Equals("VAL_T") &&
            c.Ev2.Value.Equals("VAL_P");
        cond = env.Condition(aTime, aProc, pred);
        yield return cond;
        Console.WriteLine("CUSTOM: {0}", cond.Value);
    }

    public static void Run() {
        var env = Sim.NewEnvironment();
        env.Start(CondTester(env));
        env.Run();
    }
}
```

Listato B.7 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.17.

```
using IEvents = System.Collections.Generic.IEnumerable<IEvent>;

static class TrainInterrupt {
    static readonly double AvgTravelTime;
    static readonly double BreakTime;

    static TrainInterrupt() {
        Sim.CurrentTimeUnit = TimeUnit.Minute;
        AvgTravelTime = 20.Minutes();
        BreakTime = 50.Minutes();
    }

    static IEvents Train(IEnvironment env) {
        object cause;
        string fmt;
        while (true) {
            var time = env.Random.Exponential(1.0/AvgTravelTime);
            fmt = "Treno in viaggio per {0:.00} minuti";
            Console.WriteLine(fmt, time);
            yield return env.Timeout(time);
            if (env.ActiveProcess.Interrupted(out cause))
                break;
            fmt = "Arrivo in stazione, attesa passeggeri";
            Console.WriteLine(fmt);
            yield return env.Timeout(2.Minutes());
            if (env.ActiveProcess.Interrupted(out cause))
                break;
        }
        fmt = "Al minuto {0:.00}: {1}";
        Console.WriteLine(fmt, env.Now, cause);
    }

    static IEvents EmergencyBrake(IEnvironment env,
                                  IProcess train) {
        yield return env.Timeout(BreakTime);
        train.Interrupt("FRENO EMERGENZA");
    }

    public static void Run() {
        var env = Sim.NewEnvironment(21);
        var train = env.Start(Train(env));
        env.Start(EmergencyBrake(env, train));
        env.Run();
    }
}
```

Listato B.8 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.19.

```
using IEvents = System.Collections.Generic.IEnumerable<IEvent>;

static class PublicToilet {
    static readonly double AvgPersonArrival;
    static readonly double AvgTimeInToilet;
    static readonly double SimTime;

    static PublicToilet() {
        Sim.CurrentTimeUnit = TimeUnit.Minute;
        AvgPersonArrival = 1.Minutes();
        AvgTimeInToilet = 5.Minutes();
        SimTime = 10.Minutes();
    }

    static IEvents Person(IEnvironment env,
                          string gender,
                          IResource toilet) {
        using (var req = toilet.Request()) {
            yield return req;
            Console.WriteLine("{0:0.00}: {1} --> Bagno",
                              env.Now, gender);
            var t = env.Random.Exponential(1.0/AvgTimeInToilet);
            yield return env.Timeout(t);
            Console.WriteLine("{0:0.00}: {1} <-- Bagno",
                              env.Now, gender);
        }
    }

    static IEvents PersonGenerator(IEnvironment env) {
        var womenToilet = env.NewResource(1);
        var menToilet = env.NewResource(1);
        var count = 0;
        while (true) {
            var rnd = env.Random.NextDouble();
            var gn = ((rnd<0.5) ? "Donna" : "Uomo") + count++;
            var tt = (rnd<0.5) ? womenToilet : menToilet;
            Console.WriteLine("{0:0.00}: {1} in coda",
                              env.Now, gn);
            env.Start(Person(env, gn, tt));
            var t = env.Random.Exponential(1.0/AvgPersonArrival);
            yield return env.Timeout(t);
        }
    }

    public static void Run() {
        var env = Sim.NewEnvironment(21);
        env.Start(PersonGenerator(env));
        env.Run(SimTime);
    }
}
```

Listato B.9 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.21.

```
Public Module Hospital
    Const Red = 0
    Const Yellow = 1
    Const Green = 2

    Iterator Function Person(env As IEnvironment,
                             name As String, code As Integer,
                             hospital As IResource)
        As IEnumerable(Of IEvent)
        Using req = hospital.Request(code)
            Yield req
            Console.WriteLine("{0} viene curato...", name)
            Yield env.Timeout(5)
        End Using
    End Function

    Sub Run()
        Dim env = Sim.NewEnvironment()
        Dim hospital = env.NewResource(2, WaitPolicy.Priority)
        env.Start(Person(env, "Pino", Yellow, hospital))
        env.Start(Person(env, "Gino", Green, hospital))
        env.Start(Person(env, "Nino", Green, hospital))
        env.Start(Person(env, "Dino", Yellow, hospital))
        env.Start(Person(env, "Tino", Red, hospital))
        env.Run()
    End Sub
End Module
```

Listato B.10 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.22.

```
Module HospitalPreemption
    Const Red = 0
    Const Yellow = 1
    Const Green = 2

    Iterator Function Person(env As IEnvironment,
                             code As Integer,
                             hospital As IPreemptiveResource,
                             delay As Double)
        As IEnumerable(Of IEvent)
        Yield env.Timeout(delay)
        Dim name = env.ActiveProcess.Name
        Using req = hospital.Request(code, (code = Red))
            Yield req
            Console.WriteLine("{0} viene curato...", name)
            Yield env.Timeout(7)
            Dim info As IPreemptionInfo = Nothing
            If env.ActiveProcess.Preempted(info) Then
                Console.WriteLine("{0} scavalcato da {1}",
                                  name, info.By)
            Else
                Console.WriteLine("Cure finite per {0}", name)
            End If
        End Using
    End Function

    Sub Run()
        Dim env = Sim.NewEnvironment()
        Dim hospital = env.NewPreemptiveResource(2)
        env.Start(Person(env, Yellow, hospital, 0), "Pino")
        env.Start(Person(env, Green, hospital, 0), "Gino")
        env.Start(Person(env, Green, hospital, 1), "Nino")
        env.Start(Person(env, Yellow, hospital, 1), "Dino")
        env.Start(Person(env, Red, hospital, 2), "Tino")
        env.Run()
    End Sub
End Module
```

Listato B.11 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.24.

```
Module ProducerConsumer
    Iterator Function Producer(env As IEnvironment,
                               store As IStore(Of Integer))
        As IEnumerable(Of IEvent)
        While True
            Yield env.Timeout(env.Random.Next(1, 20))
            Dim item = env.Random.Next(1, 20)
            Yield store.Put(item)
            Console.WriteLine("{0}: Prodotto un {1}",
                              env.Now, item)
        End While
    End Function

    Iterator Function Consumer(env As IEnvironment,
                               store As IStore(Of Integer))
        As IEnumerable(Of IEvent)
        While True
            Yield env.Timeout(env.Random.Next(1, 20))
            Dim getEv = store.Get()
            Yield getEv
            Console.WriteLine("{0}: Consumato un {1}",
                              env.Now, getEv.Value)
        End While
    End Function

    Sub Run()
        Dim env = Sim.NewEnvironment(21)
        Dim store = env.NewStore (Of Integer)(2)
        env.Start(Producer(env, store))
        env.Start(Producer(env, store))
        env.Start(Consumer(env, store))
        env.Run(until := 60)
    End Sub
End Module
```

Listato B.12 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.26.

```
Module ProducerFilteredConsumer
    Iterator Function Producer(env As IEnvironment,
                        store As IFilterStore(Of Integer))
        As IEnumerable(Of IEvent)
        While True
            Yield env.Timeout(env.Random.Next(1, 20))
            Dim item = env.Random.Next(1, 20)
            Yield store.Put(item)
            Console.WriteLine("{0}: Prodotto un {1}",
                              env.Now, item)
        End While
    End Function

   Iterator Function Consumer(env As IEnvironment,
                      store As IFilterStore(Of Integer),
                      name As String,
                      filter As Predicate(Of Integer))
        As IEnumerable(Of IEvent)
        While True
            Yield env.Timeout(env.Random.Next(1, 20))
            Dim getEv = store.Get(filter)
            Yield getEv
            Console.WriteLine("{0}: {1}, consumato un {2}",
                              env.Now, name, getEv.Value)
        End While
    End Function

    Sub Run()
        Dim env = Sim.NewEnvironment(21)
        Dim store = env.NewFilterStore (Of Integer)(2)
        env.Start(Producer(env, store))
        env.Start(Producer(env, store))
        env.Start(Consumer(env, store, "PARI",
                           Function(i) i Mod 2 = 0))
        env.Start(Consumer(env, store, "DISPARI",
                           Function(i) i Mod 2 = 1))
        env.Run(until := 60)
    End Sub
End Module
```

Listato B.13 Traduzione su Dessert dell'esempio per SimPy mostrato in 2.28.

```
let boxCapacity = 1.0 // Litri
let glassCapacity = 0.25 // Litri
let mutable fillBox: IEvent<int> = null

let rec filler(env: IEnvironment, box: IContainer) =
    seq<IEvent> {
    yield upcast box.Put(boxCapacity - box.Level)
    fillBox <- env.Event<int>()
    yield upcast fillBox
    let id = fillBox.Value
    printfn "%f: %d chiama tecnico" env.Now id
    yield! filler(env, box)
}

let drinker(env: IEnvironment, id, box: IContainer) =
    seq<IEvent> {
    // Occorre controllare che l'evento fillBox non sia gia'
    // stato attivato, perche' attivarlo nuovamente
    // risulterebbe in una eccezione da parte di SimPy.
    if box.Level < glassCapacity && not fillBox.Succeeded then
        fillBox.Succeed(id)
    yield upcast box.Get(glassCapacity)
    printfn "%f: %d ha bevuto!" env.Now id
}

let rec spawner(env: IEnvironment, box, nextId) =
    seq<IEvent> {
    yield upcast env.Timeout(5.0)
    env.Start(drinker(env, nextId, box)) |> ignore
    yield! spawner(env, box, nextId+1)
}

let run() =
    let env = Sim.NewEnvironment()
    let box = env.NewContainer(capacity=boxCapacity)
    env.Start(filler(env, box)) |> ignore
    env.Start(spawner(env, box, 0)) |> ignore
    env.Run(until=31)
```

Bibliografia

[1] http://en.wikipedia.org/wiki/Simula67 1

[2] http://en.wikipedia.org/wiki/Coroutine 1, 49

[3] http://en.wikipedia.org/wiki/Garbage_collection_(computer_science) 1

[4] http://www.cplusplus.com/info/history/ 1

[5] http://v.gd/8PAgWH 1

[6] http://www.arenasimulation.com/Arena_Home.aspx 1

[7] http://www.boson.com/netsim-cisco-network-simulator 1

[8] https://bitbucket.org/simpy/simpy/ 1

[9] http://www.disi.unige.it/person/RibaudoM/ 1

[10] http://www.disi.unige.it/person/ChiolaG/ 1

[11] http://en.wikipedia.org/wiki/Finite_field 1

[12] http://en.wikipedia.org/wiki/Linear_network_coding 1

[13] http://www.microsoft.com/net 2

[14] http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html 2

[15] http://www.python.org 2

[16] http://wiki.python.org/moin/GlobalInterpreterLock 2, 79, 120

[17] http://v.gd/uV4dVi 3, 4, 5, 6, 8

[18] http://v.gd/iGUObK 4, 34

[19] http://drum.lib.umd.edu/handle/1903/6294 4, 34

[20] http://v.gd/ndxgzB 4, 34

[21] http://www.physics.nus.edu.sg/∼phytaysc/articles/queue.pdf 4

[22] http://v.gd/sVOgM3 4

[23] http://teacher.buet.ac.bd/aamamun/IPE%20470-Lecture-1.pdf 4

[24] http://en.wikipedia.org/wiki/Network_simulation 4

[25] http://www.bcs.org/upload/pdf/ewic_ve09_s1paper3.pdf 4

[26] http://v.gd/7ciscJ 4

[27] http://en.wikipedia.org/wiki/Discrete_event_simulation 6

[28] http://en.wikipedia.org/wiki/Skip_list 6

[29] http://v.gd/oKjKam 6

[30] http://v.gd/z5wTuI 6

[31] http://en.wikipedia.org/wiki/Tortuga_(software) 9

[32] http://javasim.codehaus.org 9

[33] http://wiki.python.org/moin/Generators 9

[34] https://simpy.readthedocs.org/en/latest/about/history.html 11

[35] http://sourceforge.net 11

[36] http://www.simscript.com 12

[37] http://osdir.com/ml/python-simpy-user/2012-01/msg00017.html 12

[38] http://stefan.sofa-rockers.org/ 13

[39] http://v.gd/hDDC2D 13

[40] http://www.victoria.ac.nz/smsor/about/staff/tony-vignaux 13

[41] docs.python.org/2/reference/expressions.html 16

[42] http://en.wikipedia.org/wiki/Nice_%28Unix%29 31

[43] http://en.wikipedia.org/wiki/Preemption_%28computing%29 34

[44] http://en.wikipedia.org/wiki/Context_switch 34

[45] https://en.wikipedia.org/wiki/Producer-consumer_problem 36

[46] http://www.nuget.org 47

[47] http://www.microsoft.com/visualstudio/ita 47

[48] http://www.icsharpcode.net/opensource/sd/ 47

[49] http://monodevelop.com 47

[50] http://v.gd/RoqVkJ 47

[51] https://github.com/pomma89/Hippie 48

[52] https://nuget.org/packages/Hippie/ 48

[53] https://github.com/pomma89/Troschuetz.Random 48

[54] https://www.nuget.org/packages/Troschuetz.Random/ 48

[55] https://github.com/pomma89/Thrower 48

[56] https://www.nuget.org/packages/Thrower/ 48

[57] https://github.com/pomma89/Slinky 48

[58] https://www.nuget.org/packages/Slinky/ 48

[59] http://www.gnu.org/licenses/lgpl-2.1.html 48

[60] http://opensource.org/licenses/MIT 48

[61] http://www.cplusplus.com/reference/iterator/ 49

[62] http://www.chiark.greenend.org.uk/∼sgtatham/coroutines.html 49

[63] http://v.gd/8WO6yh 49

[64] http://isocpp.org/files/papers/n3708.pdf 49

[65] http://gcc.gnu.org/ 50

[66] http://gcc.gnu.org/wiki/SplitStacks 50

[67] http://jimblackler.net/blog/?p=61 50

[68] http://www.eclipse.org/aspectj/doc/next/devguide/bytecode-concepts.html 50

[69] https://code.google.com/p/infomancers-collections/ 50

[70] https://code.google.com/p/jyield/ 50

[71] https://code.google.com/p/java-yield/ 50

[72] http://code.google.com/p/coroutines/ 50

[73] http://commons.apache.org/sandbox/commons-javaflow/ 50

[74] http://v.gd/oiRNju 50

[75] http://msdn.microsoft.com/en-us/library/vstudio/hh156729.aspx 50

[76] http://msdn.microsoft.com/en-us/library/dd233209.aspx 50

[77] https://ironpython.codeplex.com/ 52, 76

[78] http://v.gd/qo3kCb 52

[79] http://v.gd/7uJZ2i 52

[80] http://www.cplusplus.com/reference/queue/priority_queue/ 52

[81] https://en.wikipedia.org/wiki/D-ary_heap 52

[82] https://en.wikipedia.org/wiki/Binomial_heap 52

[83] https://en.wikipedia.org/wiki/Fibonacci_heap 52

[84] https://en.wikipedia.org/wiki/Pairing_heap 52

[85] http://www.disi.unige.it/person/LagorioG/ 52

[86] http://www.disi.unige.it/person/AnconaD/ 52

[87] http://www.boost.org 56

[88] http://v.gd/AioC1X 56

[89] http://v.gd/Eh15u9 56

[90] http://en.wikipedia.org/wiki/Heapsort 57

[91] http://www.itu.dk/research/c5/ 59

[92] http://www.mhhe.com/engcs/compsci/sahni/enrich/c9/interval.pdf 59

[93] http://v.gd/s1uxx2 59

[94] http://msdn.microsoft.com/it-it/library/system.random.aspx 59

[95] http://docs.python.org/3/library/random.html 59

[96] http://www.codeproject.com 61

[97] http://www.codeproject.com/Articles/15102/NET-random-number-generators-and-distributions 61

[98] http://en.wikipedia.org/wiki/Categorical_distribution 61

[99] http://en.wikipedia.org/wiki/Fluent_interface 62

[100] http://msdn.microsoft.com/it-it/library/5y536ey6.aspx 62

[101] http://numerics.mathdotnet.com 62

[102] http://en.wikipedia.org/wiki/Fourier_transform 64

[103] https://github.com/mathnet/mathnet-numerics 64

[104] http://visualstudiogallery.msdn.microsoft.com/1ec7db13-3363-46c9-851f-1ce455f66970 66

[105] http://www.mono-project.com/Main_Page 67

[106] http://www.mono-project.com/Release_Notes_Mono_2.8 69

[107] http://stackoverflow.com/questions/13368134/code-contracts-in-mono 69

[108] http://v.gd/7GRaTW 69

[109] http://v.gd/EngNCs 69

[110] http://v.gd/S851eV 69

[111] http://msdn.microsoft.com/en-us/library/he2s3bh7.aspx 69

[112] https://github.com/pomma89/Dessert 73, 75, 129

[113] https://www.nuget.org/packages/Dessert 73

[114] http://msdn.microsoft.com/en-us/library/bhc3fa7f(v=vs.90).aspx 75

[115] http://v.gd/wtvLeP 75

[116] https://en.wikipedia.org/wiki/Mersenne_twister 78, 95

[117] https://en.wikipedia.org/wiki/Xorshift 79

[118] http://v.gd/jkNVzQ 79

[119] http://v.gd/IcZjSy 82

[120] https://en.wikipedia.org/wiki/Factory_method 86

[121] http://v.gd/tS9E4z 88

[122] http://msdn.microsoft.com/en-us/library/fk6t46tz.aspx 88

[123] https://en.wikipedia.org/wiki/Variadic_Templates 90

[124] http://docs.python.org/3/whatsnew/3.3.html#pep-380 98

[125] http://v.gd/XixLvI 100

[126] http://v.gd/yVHhSD 100

[127] http://v.gd/XzGvyL 103

[128] http://msdn.microsoft.com/en-us/library/dd233220.aspx 104

[129] http://v.gd/45yysw 105

[130] https://en.wikipedia.org/wiki/CPU_bound 108

[131] http://v.gd/t7W0Mv 108

[132] http://v.gd/rTc8Ic 108

[133] https://en.wikipedia.org/wiki/Erasure_code

116

Elenco degli algoritmi