ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA
CAMPUS DI CESENA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA ELETTRONICA E DELLE TELECOMUNICAZIONI PER LO SVILUPPO SOSTENIBILE

PRESTAZIONI DI PROCESSI DI MIGRAZIONE DI RETI DI MACCHINE VIRTUALI

Tesi in
Laboratorio di Reti di Telecomunicazioni LM

Relatore
Prof. Ing. Walter Cerroni

Presentata da
Laura Macrelli

Correlatore
Prof. Ing. Franco Callegati

SESSIONE II
ANNO ACCADEMICO 2012/2013

Indice

1 Introduzione **1**

2 Virtual Machine, Virtual Machine Monitor e Data Center **5**

3 Migrazione di una Virtual Machine **11**
3.1 Migrazione live e non live 12
3.2 Migrazione della memoria. 12
3.2.1 Approccio pre-copy 14
3.2.2 Approccio post-copy 16
3.2.3 Risorse Locali 18

4 Migrazione attraverso le WAN **21**
4.1 Migrazione live all'interno di una LAN 22
4.2 Requisiti per la replicazione dei dati immagazzinati 22
4.3 Requisiti di rete per la migrazione attraverso le WAN 23
4.4 Architettura per l'IP mobility 24
4.5 IP tunnel 25

5 Scenari di migrazione attraverso le WAN **27**
5.1 Migrazione senza requisiti di storage 27
5.2 Migrazione con requisiti di storage 30
5.3 Interruzioni non programmate 30

6 Virtualizzazione e tecnologie emergenti nelle Edge Network **33**
6.1 Software-Defined Networking 34
6.2 OpenFlow 36
6.3 Alcuni scenari 36

7 Prestazioni della migrazione di una singola macchina virtuale 39
7.1 Approccio analitico 40
7.1.1 Approccio analitico base 40
7.2 Approccio simulativo 43
7.3 Valutazioni quantitative dei parametri d'interesse 44
7.4 Confronto tra l'approccio analitico e l'approccio simulativo 51

8 Migrazione di insiemi di Virtual Machine 59
8.1 Migrazione di macchine virtuali in serie 60
8.1.1 Confronto tra l'approccio analitico e l'approccio simulativo. 60
8.1.2 Simulazioni di migrazioni *in serie* di reti di macchine virtuali 65
8.2 Migrazione di macchine virtuali in parallelo 68
8.2.1 Confronto tra l'approccio analitico e l'approccio simulativo. 68
8.2.2 Simulazioni di migrazioni *in parallelo* di reti di macchine virtuali 75
8.2.3 Simulazione della migrazione di macchine virtuali aventi diversa dimensione della memoria 83
8.2.4 Confronto tra i processi di migrazione 92

9 Impatto dei parametri di sistema sugli indici di prestazione 105
9.1 Valor medio e varianza di una variabile aleatoria 105
9.2 Tempo di migrazione e downtime in funzione della varianza . . . 106
9.3 Valutazioni degli indici di prestazione nel caso di parametri realistici 110

10 Caratterizzazione della *dirtying-rate* 113
10.1 Modello stocastico della gestione della memoria 115
10.2 Propagazione efficiente della memoria 117
10.2.1 Ordinamento delle pagine 118
10.2.2 Trasferimento limitato 118

11 Conclusioni **121**

Appendici **123**

Appendice A Codice del simulatore *ad eventi* per la migrazione di macchine virtuali **125**

Appendice B Codice del simulatore per la migrazione *in serie* di macchine virtuali **141**

Appendice C Codice del simulatore per la migrazione *in parallelo* di macchine virtuali **161**

Appendice D Codice del simulatore per la migrazione *in parallelo* di macchine aventi diversa dimensione della memoria **181**

Bibliografia **201**

Capitolo 1

Introduzione

Il Cloud Computing, termine ad oggi ampiamente utilizzato, è un modello per abilitare l'accesso *on-demand* ad un insieme di risorse computazionali configurabili condivise (es., reti, server, storage, applicazioni, e servizi) che possono essere fornite rapidamente, con il minimo sforzo di gestione o interazione del provider del servizio.
Questo modello di *cloud* promuove la disponibilitá, ed è composto da cinque caratteristiche essenziali, tre modelli di servizio, e quattro modelli di distribuzione [2].
Le cinque caratteristiche del Cloud Computing sono:

self-service on-demand: il consumatore puó accedere alle risorse computazionali automaticamente, in base alle proprie necessitá, senza conivolgere il provider per ogni servizio;

ampio accesso alla rete: le funzionalitá sono disponibili in rete e accessibili tramite meccanismi standard che promuovono l'utilizzo da parte di piattaforme client eterogenee;

condivisione di risorse: le risorse computazionali del provider sono messe a disposizione per servire piú consumatori possibile che utilizzano differenti risorse fisiche e virtuali assegnate dinamicamente in base alle necessitá dei consumatori stessi. Ció dá un senso di indipendenza dalla posizione in cui si trova l'utente. Le risorse includono i sistemi di archiviazione, elaborazione, memoria, la banda di rete, e le macchine virtuali;

elasticitá: le funzionalitá richieste dall'utente possono essere fornite rapidamente ed elasticamente, nella quantitá e nell'istante in cui l'utente stesso ne fa domanda;

servizio "su misura": i sistemi di Cloud controllano e ottimizzano automaticamente l'utilizzo delle risorse. In questo modo, tale utilizzo ,diventa trasparente sia al fornitore che al consumatore del servizio.

Tra i modelli di servizio troviamo:

Cloud Software as a Service (SaaS): permette al consumatore di utilizzare le applicazioni del provider in esecuzione su un'infrastruttura cloud. Le applicazioni sono accessibili da differenti dispositivi dell'utente attraverso un' interfaccia, quale puó essere un browser web. Il SaaS non consente al consumatore alcuna gestione o controllo dell'infrastruttura cloud sottostante, che include la rete, i server, i sistemi operativi (OS), ecc..

Cloud Platform as a Service (PaaS): consente al consumatore di sviluppare o acquistare applicazioni create utilizzando linguaggi di programmazione e strumenti supportati dal provider. Anche il PaaS non permette all'utente di gestire o controllare l'infrastruttura cloud, ma gli fornisce la possibilitá di controllare le configurazioni delle applicazioni sviluppate.

Cloud Infrastructure as a Service (IaaS): permette di fornire risorse computazionali fondamentali, di rete, di storage, all'utente in grado di sviluppare ed eseguire diversi software, che possono includere sistemi operativi e applicazioni. Il IaaS consente al consumatore di avere il controllo sui sistemi operativi, sui sistemi di storage, sulle applicazioni sviluppate e su alcuni componenti di rete (es., firewall).

Per quel che riguarda i quattro modelli di distribuzione, possiamo distinguere:

Private cloud: l'infrastruttura cloud è riservata esclusivamente ad un'organizzazione. Puó essere gestita dall'organizzazione stessa o da terze parti.

Community cloud: l'infrastruttura cloud è condivisa tra piú organizzazioni che hanno obiettivi comuni (es., mission, requisti di sicurezza, politiche).

Public cloud: l'infrastruttura cloud è resa disponibile al generico pubblico, o ad un ampio gruppo industriale, per esempio. In questo caso, tale infrastruttura è tipicamente posseduta un'organizzazione che vende i servizi di cloud.

Hybrid cloud: l'infrastruttura è la composizione di due o piú cloud (Private, Community, Public).

In questo nuovo paradigma, dunque, tutte le risorse sono fornite come *servizi* agli utenti finali.

Alla base del *Cloud Computing* c'è la *virtualizzazione*, tecnica che fornisce un'astrazione delle risorse hardware abilitando contemporaneamente piú istanze di sistemi operativi sulla singola macchina fisica: consente, infatti, di separare l'hardware dal software. In un sistema non virtualizzato, un singolo sistema operativo controlla tutte le risorse hardware; un sistema virtualizzato, invece, include l'hypervisor, il cui ruolo è quello di gestire gli accessi alle risorse fisiche sottostanti cosí che gli OS possano condividerle [3], e presenta all'OS stesso un insieme completo di interfacce che costituiscono una virtual machine.
Un altro vantaggio importante della virtualizzazione è la *migrazione in tempo reale*, ovvero il trasferimento di una macchina virtuale da un host fisico ad un altro, mentre questa rimane in esecuzione.
Questo tipo di migrazione è l'ingrediente chiave alla base della gestione delle attivitá dei sistemi di *Cloud Computing* per raggiungere importanti obiettivi come il *loda balacing*, il risparmio energetico, il recupero da guasti, e la manutenzione dei sistemi [1].
La tecnica piú utilizzata per la *migrazione live* prevede l'impiego dell'approccio *pre-copy*, che consiste nel trasferire tutte le pagine della memoria della macchina virtuale mentre questa continua la sua esecuzione: le pagine che si modificano vengono trasmesse e ritrasmesse finché non è raggiunto un numero massimo di *iterazioni*, o si scende sotto una pre-determinata soglia di dati modificati che rimangono da ritrasmette.
In letteratura è proposta anche la *migrazione live* che impiega l'approccio *post-copy*: prevede il trasferimento dello stato della memoria della VM solo dopo che si è trasferito lo stato della CPU, ma è meno utilizzato.
Affinché la *migrazione live* sia efficace è comunque bene valutare, caso per caso, quale strategia sia piú opportuna.
Le principali risorse fisiche impiegate nel trasferimento delle macchine *in tempo reale* sono, oltre alla memoria, la rete e il disco [4]. Mentre la memoria puó

essere copiata direttamente su un altro host, la migrazione del disco e delle interfacce di rete non è affatto banale: per mantenere aperte le connessioni ed evitare meccanismi di redirezione la VM dovrebbe mantenere il proprio indirizzo IP originale anche dopo la migrazione. Se questa avviene all'interno della stessa *local area network* (LAN), ció puó essere fatto piú facilmente rispetto a quando si considera un ambiente *wide area network* (WAN).
Della problematica della raggiungibilitá di rete e della migrazione del disco si discuterá meglio nei capitoli seguenti. Si studieranno, inoltre, altre tecnologie emergenti che, insieme alla virtualizzazione, avranno un notevole impatto, soprattutto nelle Edge Network.
Obiettivo di questa trattazione rimarranno la virtualizzazione e la migrazione delle VM: ci si concentrerá sul trasferimento della memoria e su quelle pagine che si modificano piú spesso. A tal proposito, si studierá un modello in grado di descrivere il sistema che vogliamo prendere in esame: gli indici di prestazione che ci interesseranno saranno il *tempo totale di migrazione* e il *downtime*, molto importanti quando si parla di *live migration*. Infatti, si fará riferimento a questi indici per considerare conveniente, o meno, la migrazione di una macchina virtuale all'interno di un Data Center.
S'inizierá discutendo della migrazione di una VM all'interno di una LAN, si estenderá lo studio al caso delle WAN, per poi considerare piú macchine virtuali. Si concluderá cercando di capire quali approcci sono possibili e quali scenari è opportuno considerare.

Capitolo 2

Virtual Machine, Virtual Machine Monitor e Data Center

Prima d'incominciare a parlare della migrazione di macchine virtuali all'interno di un Data Center, è bene definire il concetto di Virtual Machine, di Virtual Machine Monitor e di Data Center, per chiarezza.

Una Virtual Machine (VM) [4] è un'implementazione software di una macchina fisica su cui è installato un Sistema Operativo (OS). Tipicamente, emula la macchina fisica sottostante, ma le richieste per la CPU, per la memoria, per l'hard disk, per la rete e per le altre risorse hardware, sono gestite dal *Virtual Machine Monitor* (VMM) in esecuzione sulla macchina fisica.
Possono essere classificate, in base al loro utilizzo e al loro grado di corrispondenza con la macchina fisica sottostante, in due categorie:

1. comprende quelle macchine virtuali che forniscono una piattaforma per supportare l'esecuzione di un sistema operativo completo. Queste VM emulano l'architettura sottostante, e sono costruite sia con l'intento di fornire una piattaforma su cui eseguire i programmi lá dove l'hardware non è disponibile, sia per permettere un utilizzo piú efficiente delle risorse fisiche mantenendo il consumo di energia e i costi al minimo;

2. comprende quelle macchine virtuali progettate per l'esecuzione di un solo programma, il che significa che supportano un solo processo alla volta. Tali VM sono solitamente strettamente correlate ad uno o piú linguaggi di programmazione, e realizzate con lo scopo di fornire flessibilitá e portabilitá al programma.

Le VM possono essere facilmente migrate, copiate, riassegnate tra diversi server ottimizzando l'utilizzo delle risorse hardware, e possono eseguire diversi tipi di OS (es. Linux, Windows, ecc.) sullo stesso server fisico.
Si puó dunque riassumere che i principali benefici delle VM sono l'incapsulamento, l'isolamento, e l'indipendenza dall'hardware [5] :

l'isolamento fa si che le VM possano condividere le risorse fisiche di un singolo computer rimanendo completamente isolate le une dalle altre, come se fossero macchine fisiche distinte;

l'incapsulamento permette di considerare una VM essenzialmente come un contenitore software che incapsula tutto un insieme di risorse hardware virtuali, quali un sistema operativo e tutte le sue applicazioni. L'incapsulamento rende la VM incredibilmente portatile e facile da gestire;

l'indipendenza dall'hardware prevede che le VM siano completamente indipendenti dalla macchina fisica. Per esempio, si puó configurare una VM con componenti virtuali (es, CPU, schede di rete, ecc.) che sono completamente differenti dai componenti fisici dell'hardware sottostante. è per questo motivo che una VM puó eseguire sullo stesso server fisico OS diversi (Windows, Linux, ecc.).

D'altro canto peró, una macchina virtuale è meno efficiente di una macchina fisica in quanto accede all'hardware indirettamente.
Come precedentemente accennato, è il *Virtual Machine Monitor* che si occupa di interfacciare l'hardware e la VM.
In particolare, il VMM è il software che partiziona la macchina fisica in una, o piú, VM. Fornisce ad ogni OS un virtual BIOS, dei dispositivi virtuali, la gestione della memoria, e traduce il codice binario del kernel in nuove sequenze d'istruzioni che hanno l'effetto desiderato.
Nel progetto di un VMM è bene tenere in considerazione la *compatibilitá*, molto importante in quanto uno dei benefici chiave del VMM è la capacitá di eseguire anche software 'datati'; le *prestazioni*, intese come la misura delle spese di virtualizzazione (sono l'esecuzione della VM alla stessa velocitá con cui il software viene eseguito sulla macchina fisica); e la *semplicitá*, che è essenziale perché un guasto del VMM causerebbe l'interruzione dell'esecuzione di tutte le VM.
Come mostrato in Figura 2.1, il VMM [6] disaccoppia il software dall'hardware formando un nuovo livello tra tale software e l'hardware. Ció permette

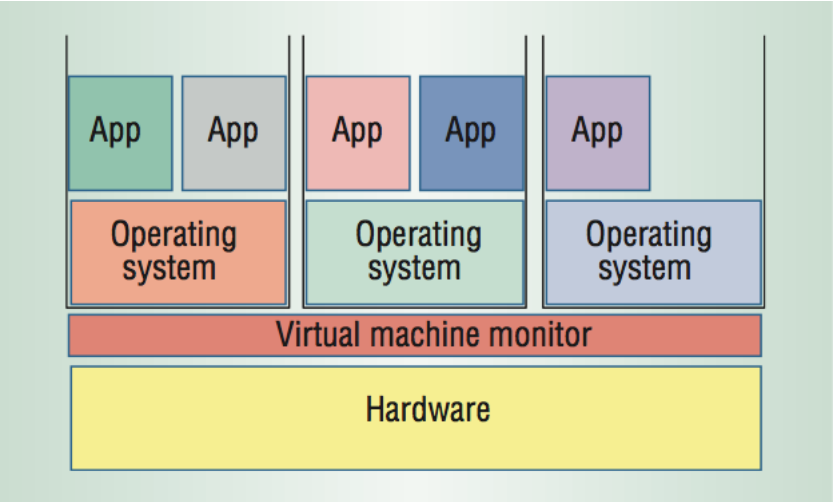


Figura 2.1: Classica architettura di un Virtual Machine Monitor.

di esercitare un controllo notevole sul modo in cui il sistema operativo utilizza le risorse hardware: ne fornisce una visione uniforme facendo apparire i dispositivi dei diversi produttori, con differenti sottosistemi I/O, identici. In questo modo, invece che preoccuparsi delle singole macchine con stingenti dipendenze tra hardware e software, gli amministratori possono vedere l'hardware come un insieme di risorse che puó eseguire differenti servizi on-demand.
In sostanza, il VMM mappa e ri-mappa le VM sull'hardware disponibile, e gestisce anche la loro migrazione.
Tra gli hypervisor piú utilizzati ricordiamo Xen, VMM open source; Hyper-V, VMM presentato dalla Microsoft; VMWare_ESX, presentato da VMWare; KVM, sviluppato dalla Red Hat Inc. [7].

La tipica architettura degli attuali Data Center [8] consiste in una topologia ad albero con due, o tre, livelli gerarchici. La topologia con due livelli puó supportare tra i 5K e gli 8K host; se si considerano piú host, invece, bisogna fare riferimento alla topologia con tre livelli gerarchici. Le foglie dell'albero sono switch con un certo numero di porte (da 48 a 288), e un certo numero di collegamenti con uno, o piú, elementi di rete che aggregano e trasferiscono i pacchetti tra gli switch foglia. Nei livelli piú alti ci sono switch con, tipicamente, da 32 a 128 porte e che hanno la capacitá di aggregare il traffico. Entrambi i tipi di switch permettono agli host di comunicare tra loro alla massima velo-

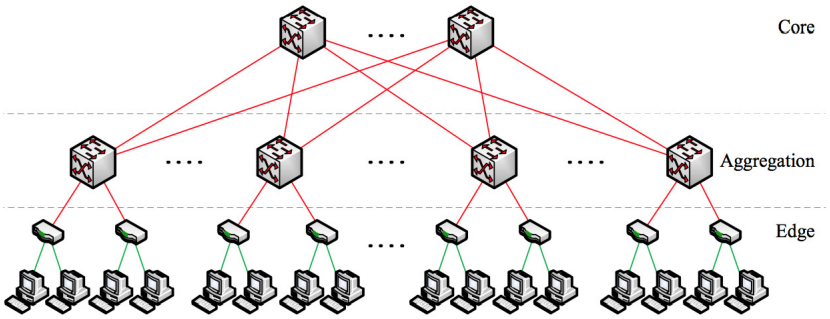


Figura 2.2: Tipica topologia di un Data Center.

citá consentita dalle loro interfacce.
Molti progetti introducono l' *oversubscription* [8] per tenere bassi i costi della struttura. Un oversubscription di 1:1 significa che gli host potrebbero comunicare con gli altri host con la massima banda permessa dalle loro interfacce di rete. Un valore di 5:1 significa che solo il 20% della banda disponibile degli host è effettivamente utilizzabile per la comunicazione. Infatti, due nodi connessi allo stesso switch fisico potrebbero comunicare alla massima banda, ma considerando diversi switch, anche posti su livelli gerarchici diversi, la banda disponibile potrebbe significativamente ridursi.
Per fronteggiare queste limitazioni solitamente s'impiegano switch e router piuttosto costosi.
È importante, dunque, che l'architettura di comunicazione di un Data Center raggiunga i seguenti obiettivi:

banda d'interconnessione scalabile: deve essere possibile, per qualsiasi host, poter comunicare con un altro host alla massima banda consentita dalle loro interfacce;

economia di scala: i personal computer devono diventare la base per un ambiente di elaborazione su larga scala, e gli switch devono rimanere economici;

compatibilitá all'indietro: l'intero sistema deve essere compatibile con gli host che eseguono Ethernet e IP. In [8] si dimostra che interconnettendo gli switch in modo da ottenere una struttura ad albero (Figura 2.2), si puó sfruttare la massima banda disponibile. Utilizzando switch comuni, gli autori hanno raggiunto costi minori di qualsiasi altra soluzione fornendo,

al tempo stesso, piú banda. La soluzione proposta non prevede alcuna modifica agli host finali, è totalmente TCP/IP compatibile, e impone leggere modifiche solo alle funzioni di forwarding degli switch stessi.

I servizi *cloud* stanno portando alla creazione di questi Data Center, che ospitano centinaia di migliaia di server, e che possono fornire un gran numero di servizi distinti (es., email, ricerche) [9]. Le motivazioni alla base della loro costruzione sono, come precedentemente accennato, sia tecniche che economiche: si fa leva sull'economia di scala per la distribuzione di massa, e si traggono i vantaggi derivanti dalla possibilitá di allocare dinamicamente i server per fronteggiare guasti, disservizi, ecc..
Per essere redditizi, i Data Center devono ottimizzare l'utilizzo delle risorse attraverso la *flessibilitá*, ovvero la capacitá di assegnare a qualsiasi server qualsiasi servizio.

Capitolo 3

Migrazione di una Virtual Machine

La migrazione è il trasferimento di una VM da un host fisico ad un altro. La *migrazione live* consente di spostare la VM mentre esegue servizi ed applicazioni in tempo reale. In questo caso, è importante che il trasferimento avvenga in modo da bilanciare il *downtime* e il *tempo totale di migrazione* [5]. Il primo è il periodo durante il quale il servizio non è piú disponibile, ed è direttamente visibile al cliente come 'interruzione del servizio'; il secondo è la durata tra l'inizio della migrazione, fino a quando la VM è perfettamente funzionante sul nuovo host fisico. A questo punto, l'host originale puó essere dismesso per manutenzione, riparazione, o aggiornamento.
È importante assicurarsi che la migrazione non degradi i servizi in esecuzione a causa della contesa delle risorse, e che sia trasparente all'OS, alle applicazioni e ai clients della VM: a tutte le parti coinvolte deve sembrare che la VM non cambi affatto la sua posizione [10].In generale, è essenziale tener conto:

- della memoria della VM;
- delle connessioni di rete, USB, ecc.,
- dello stato del disco.

3.1 Migrazione live e non live

Sapuntzakis et al. hanno descritto in [11] come migrare lo stato di un computer, che include lo stato del disco, della memoria, dei registri della CPU e dei dispositivi di I/O, attraverso una rete. Tali autori hanno chiamato questo stato *"capsula"*. La *capsula* comprende l'intero sistema operativo, le applicazioni e i processi in esecuzione quali, per esempio, Microsoft Word, Microsoft Excel, Microsoft PowerPoint e Forte (i servizi live come lo stream di un video non sono considerati in [11]).
L'architettura basata sulle capsule è chiamata "The Collective": le capsule degli utenti possono essere migrate tra casa e ufficio mentre questi sono in viaggio, possono essere duplicate, distribuite su diverse macchine e aggiornate come se fossero dei dati. Possono anche essere spostate da una macchina all'altra per bilanciarne il carico, o per guasti. Siccome le capsule non sono legate a una macchina in particolare, possono seguire gli utenti ovunque vadano.
In generale, gli autori di [11] si sono concentrati sull'ottimizzazione di una migrazione che avviene in lunghi periodi di tempo, su collegamenti lenti, e che ferma l'esecuzione del OS per tutta la durata del trasferimento.
Al contrario, per la migrazione *live*, bisogna concentrare i propri sforzi su un rapido (decine di millisecondi) trasferimento attraverso reti veloci. In questo caso, sono fattori chiave la dimensione della memoria della VM, che ha effetti sul tempo totale di migrazione e sul traffico; il tasso con cui si modificano le pagine di memoria, che influisce sul tempo totale di migrazione e sul traffico; e il tasso di trasmissione, insieme all'algoritmo per la configurazione della migrazione, cruciali per le prestazioni di migrazione.
D'ora in poi, se non diversamente specificato, si fará riferimento alla *migrazione live*.

3.2 Migrazione della memoria.

Quando una VM sta eseguendo servizi *live*, è importante che il trasferimento avvenga in modo che siano bilanciati il *downtime* e il *tempo totale di migrazione*.
Il trasferimento della memoria puó essere generalizzato in tre fasi:

Push phase: la VM sorgente continua l'esecuzione mentre certe pagine sono *spinte* attraverso la rete verso la nuova destinazione. Per mantenere la coerenza, le pagine modificate durante questo processo devono essere ritrasmesse.

Stop-and-copy phase: la VM sorgente viene arrestata, le pagine vengono copiate sulla VM destinazione, quindi viene fatta partire la VM sul nuovo host.

Pull phase: la VM viene messa in esecuzione sul nuovo host fisico e, se tenta di accedere a pagine che non sono ancora state copiate, tali pagine vengono *tirate* nella rete dalla VM sorgente.

La maggior parte delle soluzioni prevede solo una, o due, delle tre fasi.
Per esempio, una pura *stop-and-copy* coinvolge l'arresto della VM originale, la copia di tutte le pagine di memoria sulla destinazione, e poi l'avvio della nuova VM. Quest'approccio è vantaggioso in termini di semplicitá, ma comporta che entrambi *downtime* e *tempo totale di migrazione* siano proporzionali alla quantitá di memoria fisica da allocare. Ció porta ad un'interruzione inaccettabile se la VM sta eseguendo un servizio *live*.

Combinando due delle tre fasi possiamo distinguere:

l' ***approccio pre-copy*** : combina una *push phase* iterativa con una *stop-and-copy* phase molto breve. Con "iterativa" [12, 13] si intende che il pre-copy avviene in piú turni, in cui le pagine che devono essere trasferite al turno n sono quelle modificate durante il turno $n-1$. In particolare, le pagine della memoria sono trasferite dalla macchina sorgente a quella destinazione, senza fermare l'esecuzione della VM che deve migrare. Solo nell'ultima fase, quella di stop-and-copy, la VM viene messa in pausa, vengono copiate le rimanenti pagine, quindi viene fatta ripartire.
È preferibile trasferire le pagine che vengono aggiornate molto frequentemente nella fase finale.

l' ***approccio post-copy*** : una breve fase *stop-and-copy* trasferisce i dati essenziali alla destinazione, quindi la VM viene avviata sul nuovo host. Le pagine di memoria vengono trasferite al primo utilizzo, ma solo dopo che è stato trasferito lo stato della CPU. Quest'approccio assicura che le pagine siano trasferite una sola volta, evitando i costi che derivano da

molteplici trasmissioni. Il risultato è un *downtime* contenuto, ma il *tempo totale di migrazione* aumenta e le prestazioni, subito dopo la migrazione, sono inaccettabili finché non è trasferito un numero sufficientemente elevato di pagine.

3.2.1 Approccio pre-copy

In [5, 12] il processo di migrazione è descritto come interazione tra i due host coinvolti:

Fase 0: Pre-Migration: Si attiva una VM sull'host fisico A. Per velocizzare qualsiasi futura migrazione si preseleziona un host 'obiettivo' in cui sono garantite le risorse necessarie.

Fase 1: Prenotazione: È emessa la richiesta di migrazione di un OS dall'host A all'host B. Inizialmente si conferma che le risorse necessarie sono disponibili su B, quindi si "prenota" un contenitore virtuale della misura opportuna.

Fase 2: Pre-copy iterativa: Durante la prima iterazione si trasferiscono tutte le pagine da A a B. Nelle successive, si copiano solo le pagine che sono state modificate l'iterazione precedente.

Fase 3: Stop-and-copy: Si sospende l'esecuzione dell'OS su A e si ridirige il traffico su B. A questo punto si trasferiscono lo stato della CPU e le pagine di memoria modificate durante l'ultima iterazione. Alla fine di questa fase c'é la stessa copia della VM sia su A che su B. La copia su A è ancora considerata la principale e, in caso di guasti, è quella che viene riavviata.

Fase 4: Riconoscimento: L'host B indica ad A che ha ricevuto con successo un'immagine coerente dell'OS. L'host A *riconosce* tale messaggio come segno di avvenuta migrazione: ora A puó dismettere la VM e B diventa l'host principale.

Fase 5: Attivazione: Viene attivata la VM su B. Un codice opportuno ricollega i dispositivi e avverte che l'indirizzo IP ha cambiato posizione.

Con quest'approccio, quindi, una richiesta di migrazione cerca di trasferire la VM su un nuovo host e, in caso di fallimento, tale VM viene riavviata localmente, interrompendo la migrazione.
Lo svantaggio è lo spreco di risorse in cui s'incorre trasferendo piú e piú volte le pagine di memoria che si modificano con l'esecuzione della VM. A questo proposito, l'idea che hanno avuto gli autori di [12], è di trasferire durante le fasi iterative di pre-copy solo quelle pagine che si modificano raramente, trasferendo le rimanenti durante la fase di stop-and-copy.

Una pagina di memoria che viene modificata spesso è chiamata *dirty page*, e il loro insieme è il *Writable Working Set* (WWS). Tali *dirty pages* sono le pagine che includono le stack e le local variables, e le pagine usate per la rete e il disco [12, 13].
La dimensione del WWS di un sistema operativo dipende dai benchmark [14], che esso esegue: in particolare, dipende dal carico dei server (quanti clients stanno servendo) e soprattutto dalla *dirtying-rate* [13, 15].
È bene analizzare i WWS per implementare un pre-copy efficiente e controllabile.

Dynamic rate-limiting

Per quel che riguarda il traffico di migrazione, non sempre è conveniente selezionare un singolo limite di banda. Nonostante un limite basso evita di influire negativamente sulle prestazioni dei servizi in esecuzione sprecando banda e CPU, provocherebbe l'aumento del *downtime*, come dimostrato in [12] . Considerando l'approccio *pre-copy*, la soluzione è di adattare dinamicamente il limite di banda ad ogni iterazione.
L'amministratore seleziona un limite massimo e uno minimo: alla prima iterazione le pagine vengono trasferite alla banda minima, poi ogni iterazione successiva conta il numero di pagine che si erano modificate l'iterazione precedente, divide questo numero per la durata di tale iterazione, e calcola cosí la *dirtying-rate*. Il limite di banda per il turno successivo è determinato aggiungendo una costante incrementale alla *dirtying-rate* dell'iterazione precedente.
I turni di pre-copy terminano quando la *dirtying-rate* calcolata è maggiore del limite massimo, o quando rimangono pochi KB da trasferire.
Durante la fase finale di stop-and-copy si minimizza il *downtime* trasferendo la memoria al massimo rate possibile.
Sfruttando questo schema adattativo, l'utilizzo di banda rimane basso durante

il trasferimento della maggior parte delle pagine, aumentando solo alla fine della migrazione per il trasferimento delle *dirty pages* del WWS.
Per tenere traccia degli accessi alla memoria è possibile impiegare l'hypervisor (es. Xen, VMware ESX, Hyper-V hanno tutti questa capacitá) utilizzando un meccanismo chiamato *shadow page tables*. L'hypervisor mantiene queste *shadow page tables* e traduce le *page tables* delle VM: é, infatti, in grado di "intrappolare" tutti gli aggiornamenti di memoria di una VM e mantenere una *bitmap* in cui marca le pagine modificate.

Dirty pages

Ogni OS ha un certo numero di pagine che sono aggiornate molto frequentemente, e che quindi non sono delle buone candidate per la migrazione *pre-copy*, nemmeno sfruttando tutta la banda disponibile. Gli autori di [12, 13, 16, 17, 18] hanno osservato che le pagine che si modificano rapidamente sono quelle che tenderanno a modificarsi piú spesso. Decidono quindi di guardare periodicamente la *dirty bitmap* e di trasferire solo quelle pagine che si sono modificate il turno precedente, ma che non sono modificate quando si guarda la *bitmap*.

3.2.2 Approccio post-copy

Con questo metodo, prima si trasmette lo stato del processore, si attiva la VM sul nuovo host, poi si trasferiscono le pagine di memoria.

Prevede [19]:

Fase 0: Tempo di Preparazione: È il periodo tra l'inizio della migrazione e il trasferimento dello stato del processore della VM all'host destinazione. Durante questo periodo la VM continua l'esecuzione e modifica le pagine della memoria.

Fase 1: Downtime: È il tempo durante il quale viene fermata l'esecuzione della VM. Durante questo periodo è trasferito lo stato della CPU, oltre al minimo indispensabile per far ripartire la VM sul nuovo host.

Fase 2: Tempo di Ripresa: È il periodo tra l'attivazione della VM sul nuovo host e la fine della migrazione. La maggior parte dell'approccio *post-copy* opera in questa fase durante la quale vengono trasferite le pagine di memoria.

Per trasferire le pagine di memoria da sorgente a destinazione gli autori di [19] adottano la combinazione di quattro tecniche: *demand-paging, active push, prepaging, e dynamic self-ballooning* (DSB).
La demand-paging assicura che ogni pagina sia spedita sulla rete solo una volta. Analogamente, l'active push assicura che siano rimosse le dipendenze residue [12] dalla sorgente piú in fretta possibile. La prepaging utilizza gli accessi alle pagine della VM come "suggerimenti" per ridurre il numero di richieste esplicite delle pagine di memoria (network faults) e la durata della Fase 2. La tecnica DSB riduce il numero di pagine *libere*, non allocate, che vengono trasferite durante una migrazione.
Gli obiettivi dell'approccio *post-copy via prepaging* sono anticipare i faults e adattare il trasferimento delle pagine in modo da riflettere gli accessi alla memoria.
Siccome è impossibile predirre esattamente tali faults, l'idea degli autori di [19] è di utilizzarli come suggerimenti per stimare la posizione nello spazio dei futuri accessi alla memoria. L'efficacia della strategia di prepaging è tanto maggiore quanto è minore il numero di richieste esplicite delle pagine di memoria.
A tal proposito, è importante riuscire a trovare un criterio che permette di capire quali pagine saranno richieste esplicitamente nel prossimo futuro, e inviarle prima che la VM ne abbia bisogno. Gli autori propongono:

***bubbling con singolo punto di partenza*:** le pagine di memoria della VM sono mantenute in un dispositivo chiamato *in-memory pseudo-paging*. Il componente per l'active push si posiziona sul punto di partenza, detto *pivot*, e trasmette in maniera simmetrica le pagine da una parte e dall'altra di tale pivot ad ogni iterazione.
L'algoritmo è chiamato *bubbling* proprio perché è simile ad una bolla che cresce attorno al pivot centrale. Il pivot è inizializzato alla prima pagina del dispositivo *in-memory pseudo-paging*, il che significa che la bolla si espanderá solo in una direzione. In seguito, quando avviene una richiesta esplicita di una pagina, un apposito componente sposta il pivot nella nuova posizione e si comincia a far crescere una nuova bolla, questa volta in entrambe le direzione.

***bubbling con piú punti*:** questa tecnica considera il caso in cui la VM ha piú processi in esecuzione contemporaneamente. Un singolo pivot non

sarebbe sufficiente a ridurre le richieste esplicite delle pagine di memoria, quindi si estende il *bubbling* precedente considerando piú pivot.

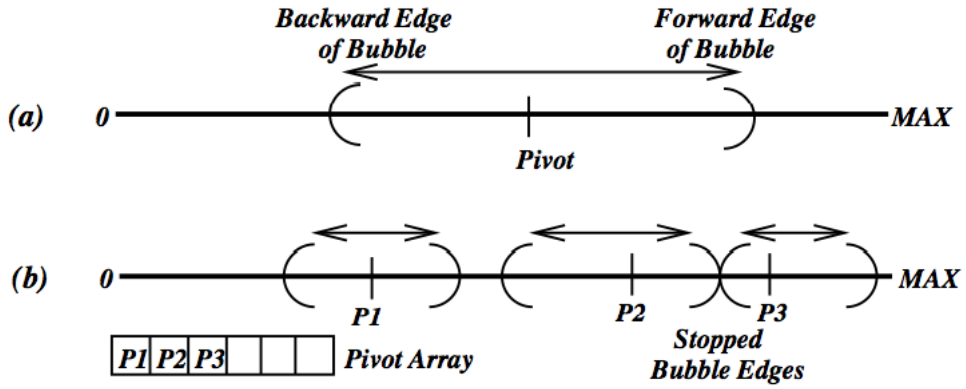


Figura 3.1: a) Bubbling con singolo pivot. b) Bubbling con piú pivot

La tecnica *dynamic self-ballooning* (DSB) è presentata in [19] per migliorare le prestazioni sia dell'approccio post-copy, sia del pre-copy: prima che inizi la migrazione, una VM ha un certo numero di pagine di memoria libere, non allocate. Trasferire queste pagine vorrebbe dire sprecare risorse di rete e CPU, e aumentare il tempo totale impiegato per la migrazione, qualsiasi approccio si scelga. Per questi motivi si vuole evitare di trasmettere tali pagine.
Il *ballooning* [20] permette di ridimensionare l'allocazione della memoria di una VM: riduce il numero di pagine libere senza incidere sull'esecuzione della VM, cosí che quest'ultima possa essere migrata il piú velocemente possibile.

3.2.3 Risorse Locali

Mentre la memoria puó essere copiata direttamente sulla destinazione, le connessioni ai dispositivi locali, come dischi e interfacce di rete, richiedono maggiori considerazioni. Per quanto riguarda le risorse di rete, è preferibile che l'OS che deve migrare mantenga aperte tutte le connessioni.
A questo proposito, la VM dovrá includere tutti i protocolli di stato (es.: TCP PCB) e dovrá portarsi dietro il proprio indirizzo IP. In [12] è stato osservato che, all'interno di un Data Center, le interfacce di rete delle macchine sorgente

e destinazione sono tipicamente sulla stessa switched LAN [21]. In questo caso, la soluzione degli autori, è di far generare alla macchina migrata un *unsolicited ARP reply* che avverte gli altri dispositivi che quell'indirizzo IP, d'ora in poi, si troverá in un'altra posizione.
La migrazione dello *storage* [22] puó essere affrontata in maniera analoga: i Data Center piú moderni utilizzano dispositivi *network-attached storage* (NAS) per mantenere i dati. Il NAS ha diversi vantaggi, tra cui un'amministrazione semplice e centralizzata, e un'accessibilitá uniforme per tutti gli host del Data Center.

Capitolo 4

Migrazione attraverso le WAN

In ambiente LAN, le nuove tecnologie di virtualizzazione [10, 12] hanno dimostrato di essere mezzi efficienti per la gestione di un Data Center: non solo permettono eventi di manutenzione, ma possono anche essere utilizzate per bilanciare il carico dei server.
Quando si è all'interno di una LAN, finché il server virtuale mantiene il suo indirizzo IP, ogni interconnessione rimane attiva. Analogamente, gli eventuali problemi per lo storage sono risolti con un NAS che rimane raggiungibile da qualsiasi server della rete.
Sfortunatamente, la migrazione di uno o piú server in una WAN non è altrettanto semplice per due ragioni:

- la prima è che la migrazione live richiede che il server virtuale mantenga lo stesso indirizzo di rete cosí che la connettivitá del server migrato sia indistinguibile da quella del server originale;

- la seconda è che i meccanismi sviluppati per il *disaster recovery* sono poco adatti alla migrazione live in quanto, in generale, le tecnologie disponibili non sono 'consapevoli' delle applicazioni.

Per migrare servizi e applicazioni da un Data Center ad un altro bisogna considerare tre sottosistemi essenziali : i server, la rete, e lo storage.
La soluzione proposta in [23] prevede la cooperazione di tutti e tre i sottosistemi in modo da poter migrare i servizi real-time senza interruzioni significative.

4.1 Migrazione live all'interno di una LAN

Come descritto in precedenza, è possibile trasferire un OS, in esecuzione come server virtuale per esempio, da una macchina fisica ad un'altra.
Assumendo sia possibile raggiungere gli indirizzi di rete del server virtuale, bisogna che le applicazioni e le connessioni in corso rimangano attive. Per mantenere tale raggiungibilitá di rete, gli indirizzi IP associati al server devono essere raggiungibili anche sul nuovo host fisico.
In ambiente LAN, questo è possibile mediante l'emissione di un *unsolicited ARP reply* che crea un collegamento tra il nuovo indirizzo MAC e l'indirizzo IP, o facendo affidamento sulle tecnologie di livello-due affinché permettano al server virtuale di riutilizzare il suo (vecchio) MAC.
Per quando riguarda il *disk storage*, se si considera un NAS per immagazzinare i dati, allora diventa solo un'altra applicazione di rete e perció puó essere gestita con la migrazione basata su LAN.
In ambiente non-LAN fare tutto ció diventa complicato.

4.2 Requisiti per la replicazione dei dati immagazzinati

Affinché i dati siano sempre disponibili, tipicamente si decide di replicarli in sistemi di *storage* remoti ai quali è sempre possibile accedere.
Tale replicazione puó essere classificata in due categorie:

replicazione sincrona : ogni dato scritto nel sistema di *storage* locale è replicato nel sistema remoto prima che avvenga un'altra scrittura;

replicazione asincrona : in questo caso i sistemi locale e remoto possono divergere. L'ammontare della divergenza è tipicamente limitata ad una certa quantitá di dati, o da un certo tempo.

La *replicazione sincrona* è solitamente raccomandata per applicazioni, come database finanziari, dove la coerenza tra il sistema per la raccolta dei dati locale e remoto ha la prioritá. Tuttavia, questa desiderabile proprietá, ha un prezzo sia in termini di performance delle applicazioni, sia sulla quantitá di lavoro che un server puó fare in un certo tempo (*throughput*), sia sul tempo totale di migrazione.

L'alternativa è utilizzare la *replicazione asincrona*.
Siccome tale soluzione permette una certa divergenza tra ció che è scritto nel sistema locale e in quello remoto, alcuni dati potrebbero andare persi in caso di guasto al sistema primario. Ma, grazie al fatto che le operazioni di scrittura sono limitate, la replicazione asincrona è piú efficiente nello spostare i dati da una parte all'altra della rete.
In [23] è proposta una replicazione che parte asincrona e termina sincrona: per trasferire la maggior parte dei dati tra i Data Center coinvolti conviene utilizzare la replicazione asincrona e beneficiare della sua efficienza. Quando la parte piú consistente dei dati è stata trasferita, si passa alla replicazione sincrona per completare la migrazione.
In questo modo, quando il server virtuale sará avviato nel nuovo Data Center, i requisiti di storage saranno soddisfatti.

4.3 Requisiti di rete per la migrazione attraverso le WAN

Il requisito di rete chiave per la migrazione attraverso le WAN è la raggiungibilitá dell'indirizzo IP del server virtuale alla nuova locazione subito dopo la migrazione. Bisogna peró tenere conto che l'IP *mobility* è una questione aperta da anni ma ancora problematica; che gli attuali protocolli di routing hanno problemi di convergenza dunque sono poco adatti visti i tempi stringenti della migrazione live; e che, ad oggi, la connettivitá delle WAN è amministrata e controllata esclusivamente dagli Operatori di rete.
Gli autori di [23] propongono una loro soluzione supponendo che non tutti i cambiamenti della rete debbano avvenire in tempi brevi. In particolare, permettono al software che gestisce la migrazione di cominciare ad apportare le modifiche necessarie appena si accorge che ne avverrá una.
Per collegare preventivamente i Data Center che potrebbero essere interessati alla migrazione gli autori decidono di fare uso dell' *IP tunnel* [24, 25]. Una volta che la migrazione è completata, il software che la gestisce inizia a dirigere il traffico verso il tunnel.

4.4 Architettura per l'IP mobility

In [26] è proposta un'architettura per l' *IP mobility* chiamata MAT: Mobility Support Architecture and Technologies. È MAT che consente la comunicazione tra gli host finali.
Per poterla adottare è indispensabile che tutti i nodi includano le funzioni MAT.
Ogni Virtual Machine 'mobile' è chiamata Mobile Node (MN), e ogni Virtual Machine che comunica con un MN è chiamata Correspondent Node (CN).
Un MN utilizza due indirizzi IP: l'Home Address (HoA) e il Mobile Address (MoA). L'HoA è l'indirizzo IPv6 utilizzato permanentemente da un'applicazione, il MoA invece è un indirizzo IPv6 temporaneo, che cambia ogni volta che un MN migra. Dopo che è stato allocato il MoA, il MN aggiorna l'IP Address Mapping Server (IMS) per assicurare la consistenza tra HoA e MoA.
Il kernel traduce l'indirizzo IP sorgente HoA (MoA) nell'appropriato indirizzo IP MoA (HoA): in questo modo, il mapping tra i due indirizzi è gestito dall'IMS, e la raggiungibilitá dei pacchetti è garantita dal MN stesso attraverso il kernel MAT.
Questo metodo potrá portare l'host sorgente a configurare la rete prima della migrazione: si configurano MoA e *default gateway* cosí che l'IP *mobility* consista solamente nel cambiare l'interfaccia con il MoA dopo la migrazione.
Andando piú in dettaglio, questo meccanismo prevede quattro step: il primo è l'impostazione del *default gateway*, il secondo è l'allocazione del MoA, il terzo è il cambio delle connessione e il quarto è il cambio dell'interfaccia primaria. In Xen, utilizzato in [26] come tecnologia di virtualizzazione, l'OS che controlla la VM è chiamato *Domain 0* (Dom0), e l'OS della VM è chiamato *Domain U* (DomU). Per controllare la migrazione, gli autori propongono un Migration Control Server (MCS) che gestisce le informazioni necessarie per le configurazioni di rete quali: l'indirizzo IP dell'host, il nome del dominio della VM e le informazione sul *default gateway*.
Al primo step la sorgente esegue la configurazione del *default gateway* (GW): il Dom0 raccoglie informazioni sul GW destinazione, compresi l'indirizzo IPv6 e il link local address, imposta gli indirizzi sulla seconda interfaccia di rete momentaneamente scollegata, richiede l'aggiornamento dell'IP table, e che venga attivata la seconda interfaccia presentando il *link local address* al DomU.
Nel secondo step, il DomU configura il MoA: prima di tutto conferma che l'interfaccia primaria è gestita dal DomU stesso, imposta un *link local address*

fittizzio nell'IP table come interfaccia secondaria del GW e poi attiva l'interfaccia secondaria con la configurazione del MoA. Alla fine, il DomU comunica al Dom0 che l'IP table è stata aggiornata e che il MoA è stato allocato.
A questo punto, la configurazione di rete è conclusa e si attende l'inizio della migrazione. Una volta che quest'ultima è terminata, si prosegue con gli step successivi.
Nel terzo passo, il Dom0 della destinazione assicura la raggiungibilitá alle applicazioni in esecuzione su una VM cambiando i collegamenti dell'interfaccia virtuale: precisamente, dopo la migrazione, il Dom0 collega l'interfaccia secondaria al virtual bridge scollegando la primaria.
Nel quarto step, il Dom0 nomina primaria l'interfaccia che era stata, fino a quel momento, la secondaria. In questo modo, il server situato nel nuovo data center è subito raggiungibile dal client.

L'interrogativo che rimane studiando quest'approccio è capire come il client apprende l'indirizzo IP MoA del MN, smettendo quindi di inviare i pacchetti alla vecchia destinazione.

4.5 IP tunnel

La migrazione a lungo raggio richiede meccanismi di strato-tre [27] per mantenere la connettivitá della VM. Quest'ultima acquisisce l'indirizzo IP dalla rete in cui è in esecuzione, cosa che puó rompere le sessioni TCP attive con i clients, dunque è necessario che mantenga lo stesso indirizzo IP prima, durante, e dopo la migrazione. Affinché sia possibile, si sfrutta la configurazione dinamica di un IP *tunnel* che consente di mantenere la connettivitá con i clients: finché l'IP rimane lo stesso, le riconfigurazioni saranno invisibili al TCP, e a qualsiasi altra sessione di livello superiore.
La riconfigurazione di un IP *tunnel*, proposta in [25] e riportata in Figura 4.1, mostra uno scenario composto di due distinti siti tra cui far migrare una virtual machine.
I due siti sono dotati di indirizzi di rete diversi e globali (12.x.x.x e 13.x.x.x). Alla macchina virtuale è assegnato un indirizzo fittizio (10.1.1.2) e una "virtual gateway interface" (10.1.1.1), creata dall'hypervisor, attraverso la quale comunica con il mondo esterno. L'elemento etichettato come "Visualization host" indica il client che è in esecuzione sulla macchina virtuale e che potrebbe

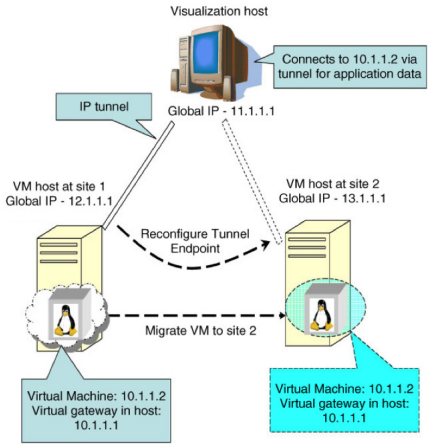


Figura 4.1: Migrazione con IP tunnel

essere situato in una terza locazione che nulla ha a che fare con i due siti che ospitano la VM.

Capitolo 5

Scenari di migrazione attraverso le WAN

Potrebbe essere necessario migrare un intero Data Center attraverso le WAN per un intervento di manutenzione programmata, o per un guasto improvviso. In [23] sono riportati diversi casi studio: nel primo, il servizio non ha requisiti di storage; nel secondo, il servizio richiede anche la migrazione dei dati immagazzinati; nel terzo è descritta la migrazione nel caso di guasto improvviso.

5.1 Migrazione senza requisiti di storage

Se non è necessaria la replicazione dei dati, gli elementi primari che dobbiamo coordinare sono la migrazione del server e la "mobilitá" della rete.
Prima della manutenzione, il Migration Management System (MMS) dovrá segnalare sia al sistema di gestione dei server, sia al sistema di gestione della rete, che è imminente una migrazione.

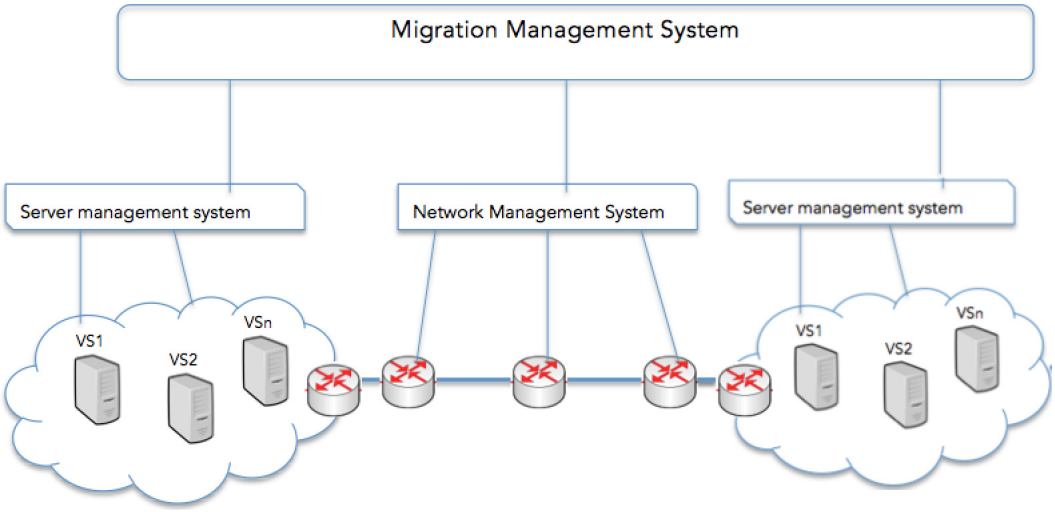


Figura 5.1: MMS e sistemi di gestione dei server e della rete

Con riferimento a Figura 5.2, il sistema di gestione dei server dará inizio al trasferimento del server virtuale dall'host fisico A (PSa) all'host fisico B (PSb). Dopo aver inizialmente trasferito la parte consistente dello stato del server vir-

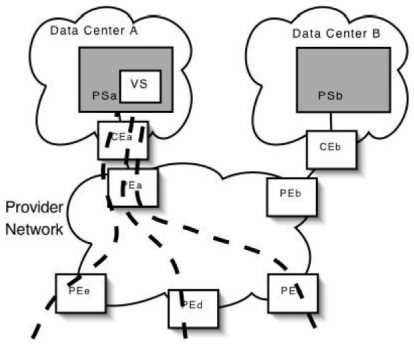


Figura 5.2: Stato iniziale della migrazione attraverso le WAN

tuale (VS), il sistema di gestione del server copia ogni cambiamento di stato da un server all'altro.
Analogamente, per quanto riguarda il sistema di gestione della rete, basandosi sulla segnalazione ricevuta dal MMS, il router *service provider edge* (PE) comincia a preparare la migrazione. In particolare, viene creato un tunnel tra

PEa e PEb (Figura 5.3) che sará utilizzato successivamente per trasferire i dati destinati al VS verso il Data Center B. Quando il MMS lo ritiene opportuno, viene interrotta l'esecuzione del VS e viene inviato un secondo segnale al sistema di gestione dei server e della rete. Per il primo sistema, questo segnale indica la fine della migrazione del VS dal Data Center A al Data Center B. Per il secondo, invece, indica che da quel momento in poi il traffico destinato all'indirizzo del VS che arriva al PEa deve essere immediatamente mandato, attraverso il tunnel, al PEb che lo consegna al Data Center B.

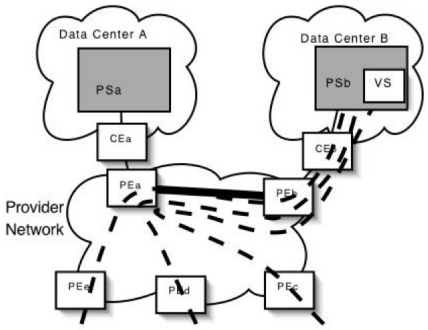


Figura 5.3: Tunnel tra PEa e PEb

Notiamo che, a questo punto, dalla prospettiva del server la migrazione è terminata. È vero peró che per arrivare al Data Center B il traffico deve passare prima per PEa, quindi attraverso il tunnel, poi per PEb. Per correggere questa situazione è necessario un altro accorgimento che coinvolge la rete. Precisamente, PEb avverte gli altri router che è possibile raggiungerlo da percorsi migliori di quello esistente. In questo modo, tali router, possono iniziare a mandargli il traffico direttamente, senza passare per PEa (Figura 5.4).

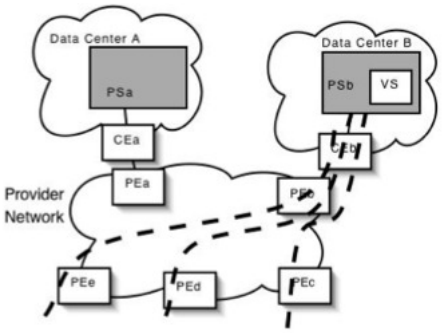


Figura 5.4: Percorso del traffico al termine della migrazione

5.2 Migrazione con requisiti di storage

Quando devono essere replicati anche i dati, è importante bilanciare le prestazioni e i *recovery point*. Per provarci, si utilizza la *replicazione asincrona* prima che cominci la migrazione.
Come sappiamo, la replicazione asincrona permette di minimizzare l'impatto sulle prestazioni delle applicazioni. Tuttavia, quando la manutenzione è imminente, il MMS dovrá comunicare al sistema di replicazione di modificare la modalitá di replicazione in *sincrona* in modo da assicurare la coerenza dei dati, anche se questo porterá ad una riduzione delle prestazioni.
Quando il MMS segnala al sistema incaricato di immagazzinare i dati il cambiamento di replicazione, lo *storage* completa tutte le operazioni asincrone in corso, e procede per trasformare le successive scritture in sincrone. A questo punto, il VS è reso inattivo e viene segnalato alla rete di trasferire il traffico verso il nuovo Data Center. Da questo momento, sia lo *storage* sia le operazioni di migrazione sono attivati nel Data Center B.

5.3 Interruzioni non programmate

Per affrontare interruzioni non programmate, in [23] è proposta una migrazione "consapevole" e cooperativa.
Per recuperare il controllo e lo stato delle applicazioni, gli autori propongono di utilizzare dei *checkpoint* che permettono di salvare periodicamente lo stato dell'applicazione.
In caso di guasti, o malfunzionamenti improvvisi, alcuni dati potrebbero anda-

re persi, ma se i *checkpoint* sono molto frequenti, il recupero puó essere efficace e abbastanza veloce.
E' importante anche la gestione della rete: in tal caso si puó procedere come descritto nel paragrafo "Migrazione senza requisiti di storage".
Mantenere lo stato dell'applicazione, o addirittura una replica virtuale dell'intera applicazione, ha peró un costo in termini di utilizzo delle risorse, anche se permette il recupero immediato in caso di guasti.
Per un buon progetto sará necessario trovare il giusto compromesso tra tali parametri.

Capitolo 6

Virtualizzazione e tecnologie emergenti nelle Edge Network

Si è parlato di virtualizzazione, di migrazione di macchine virtuali attraverso LAN, poi attraverso WAN. A questo punto puó essere interessante analizzarne qualche impiego.

Tecnologie emergenti come il Software Defined Networking (SDN) [28] e la virtualizzazione potrebbero aiutare gli Operatori di rete nella gestione e nella configurazione delle attrezzature, nella riduzione dei costi, nel limitare gli errori umani, ecc. [29]. Nelle Edge Network tali tecnologie e soluzioni potrebbero portare ad un'importante innovazione, grazie anche ai progressi fatti con le tecnologie embedded di elaborazione e comunicazione.

Figura 6.1: Core e Edge Network

Una delle sfide principali alla base di questa visione è la capacitá di istanziare e orchestrare dinamicamente le VM che eseguono servizi di rete attraverso le reti dei providers.

Il SDN si propone di disaccoppiare la logica del piano di controllo dall'hardware, e di trasferire il controllo dello stato della rete ad un componente chiamato "controller". Anche la virtualizzazione delle risorse fisiche ha un notevole impatto sull'evoluzione della rete.
Come risultato, gli Operatori possono istanziare dinamicamente, attivare, riallocare e programmare funzioni e risorse in accordo con le necessitá e le politiche dell'Operatore stesso.
Per affrontare l'incremento di complessitá nel gestire e controllare la rete, gli Operatori richiedono, oltre alla virtualizzazione, l'integrazione di capacitá cognitive, cioé che la rete stessa sia in grado di imparare dall'ambiente circostante e, sulla base delle informazioni ottenute, sia in grado di migliorare le prestazioni.

6.1 Software-Defined Networking

Nell'architettura Software-Defined Networking (SDN) il piano del controllo e dei dati sono disaccoppiati, l'intelligenza e lo stato della rete vengono centralizzati e l'infrastruttura di rete sottostante è indipendente dalle applicazioni. In questo modo, gli Operatori e gli utenti guadagnano in programmabilitá e controllo delle rete, favorendo la progettazione di una struttura scalabile, flessibile, e in grado di adattarsi ai cambiamenti guidati dal mercato.
Per consentire la comunicazione tra il piano del controllo e quello dei dati, SDN utilizza il protocollo *OpenFlow*. Tale protocollo è la prima interfaccia standard, progettata esattamente per il SDN, che permette di ottenere elevante prestazioni e di avere un attento controllo sul traffico, anche attraverso dispositivi di rete di diversi produttori.

Il SDN basato su OpenFlow è attualmente impiegato in numerosi dispositivi e software di rete, portando benefici sia agli utenti che agli Operatori, quali:

- la gestione centralizzata e il controllo dei dispositivi di rete di distinti produttori;

- l'automazione e la gestione attraverso le API cosí da svincolarsi dai dettagli dell'infrastruttura di rete sottostante;

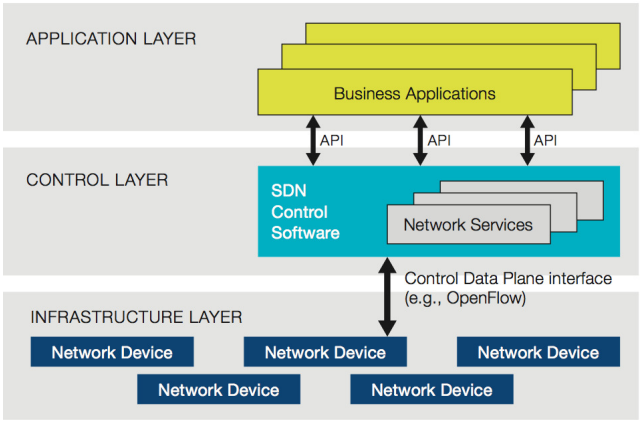


Figura 6.2: Architettura SDN

- una rapida innovazione grazie alla capacitá di eseguire nuovi servizi di rete senza il bisogno di configurare ogni singolo dispositivo o di aspettare le licenze dei produttori;

- l'aumento dell'affidabilitá e della sicurezza di rete grazie al controllo centralizzato dei dispositivi.

L'incremento di dispositivi mobili, la virtualizzazione dei server e l'avvento di servizi *cloud* portano, infatti, a dover riesaminare le tradizionali architetture di rete. Molte delle reti attuali sono gerarchiche, costruite con switch Ethernet organizzati in strutture ad albero. Tale progetto ha senso quando il traffico è principalmente tra un server e un utente, ma è poco adatto ai bisogni dinamici dei Data Center di oggi: le applicazioni accedono a differenti server e database, e gli utenti accedono alla rete con qualsiasi tipo di dispositivo, connettendosi da qualsiasi luogo, a qualsiasi ora. I gravi limiti delle attuali architetture sono, quindi, la complessitá e le "politiche incoerenti".
Negli ultimi anni, per soddisfare i bisogni tecnologici e di mercato, l'industria ha sviluppato protocolli di rete che permettono di raggiungere elevate prestazioni e affidabilitá, una connettivitá piú ampia, e una sicurezza piú stringente. Tali protocolli risolvono uno specifico problema senza il beneficio di alcuna astrazione. Per esempio, per aggiungere o spostare un dispositivo, è necessario passare per diversi switch, router, firewall, autentificazioni Web, ecc. e aggiornare la QoS e altri meccanismi che interessano il protocollo. Per far tutto ció bisogna tenere conto del modello di switch del produttore, della versione del

software, e della topologia di rete. Proprio a causa di questa complessitá le reti di oggi sono tanto piú statiche quanto piú si cerca di minimizzare il rischio d'interruzione dei servizi. Inoltre, per implementare una politica di rete generale, è necessario configurare centinaia di dispositivi: per esempio, ogni volta che si vuole attivare una nuova VM bisogna riconfigurare gli *access control list* (ACL) dell'intera struttura.
L'architettura SDN, invece, rende possibile gestire la rete attraverso le API che ne forniscono un'astrazione, permettendo l'allocazione on-demand delle risorse, di avere una rete davvero virtualizzata, servizi *cloud* e maggiore sicurezza. Inoltre, grazie ad OpenFlow, con il SDN è risolto anche il problema di gestire e programmare i dispositivi dei diversi produttori.

6.2 OpenFlow

OpenFlow [28] è la prima interfaccia standard di comunicazione tra il piano del controllo e del forwarding in un'architettura SDN. Consente l'accesso diretto e il controllo dei dispositivi di rete come switch e router, sia fisici che virtuali. E' fondamentale per portare il controllo della rete fuori dai dispositivi, permettendone la gestione centralizzata via software.
OpenFlow specifica primitive che possono essere utilizzate da applicazioni esterne per programmare il forwarding dei dispositivi. Deve essere implementato su entrambi i lati dell'interfaccia tra l'infrastruttura di rete e il software di controllo del SDN, e sfrutta il concetto di flusso per identificare il traffico basandosi su regole decise staticamente, o dinamicamente. Questo permette all'architettura SDN di controllare il traffico in maniera attenta e minuziosa.

6.3 Alcuni scenari

Di seguito si riportano scenari che coinvolgono l'architettura SDN e la virtualizzazione che potrebbero interessare gli Operatori di rete:

Aggiornamento delle reti : i servizi e le funzioni di rete potrebbero essere inizialmente sviluppate e testate in un certo ambiente. In un secondo momento, tali servizi e funzioni, potrebbero essere trasferiti in un altro ambiente (es. in un altro dominio o in un'altra Nazione) utilizzando magari risorse fisiche e di elaborazione situate altrove.

Federazione di reti : differenti reti virtuali possono essere confederate, anche se sono geograficamente remote. Questo è lo scenario ideale per un Operatore che vorrebbe fornire i suoi servizi anche attraverso differenti domini e Nazioni.

Partizione delle reti : è lo scenario contrario al precedente. Una rete virtuale, che fornisce determinati servizi, puó essere partizionata in sotto-reti piú piccole in modo da semplificarne l'amministrazione, per esempio.

Sono le Edge Network i segmenti di rete in cui le soluzioni discusse sopra hanno il maggior impatto.

Capitolo 7

Prestazioni della migrazione di una singola macchina virtuale

Concentriamoci, d'ora in poi, nello studio della migrazione di macchine virtuali all'interno di un Data Center.
Si vuole valutare quale macchina virtuale puó essere migrata a costo minimo. A tale proposito, si presta particolare attenzione al *tempo totale di migrazione*, e al *downtime* di una VM, in quanto sono proprio questi gli indici che permettono di misurare l'impatto che la migrazione ha sul consumo delle risorse di rete, e sulla disponibiliá dei servizi che interessano gli utilizzatori.
Cerchiamo,allora, di capire come valutare gli indici di prestazione.

Per valutare le prestazioni di un sistema esistono due approcci:

l'approccio analitico: richiede un modello matematico in grado di descrivere il sistema reale;

l'approccio simulativo: permette di descrivere un sistema a qualsiasi livello di dettaglio, facilitando cosí la costruzione del modello. È piú adatto allo studio di sistemi complessi.

Per entrambi gli approcci è richiesto il *modello del sistema* che mette in relazione i parametri d'ingresso con gli indici di prestazione che si vogliono valutare.

Noi vogliamo, in una prima fase, concentrare lo studio su una singola migrazione alla volta: tutte richieste di migrazione successive alla prima andranno

in coda. Supponiamo, inoltre, che tali *eventi* vengano serviti seguendo l'ordine di arrivo delle richieste.

7.1 Approccio analitico

Cerchiamo di determinare quale VM potrebbe essere migrata a costo minimo individuando il *modello matematico* che meglio descrive il sistema, e che permette di valutarne le prestazioni.

7.1.1 Approccio analitico base

È valido quando la *dirtying-rate* rimane la stessa. Tale *dirtying-rate* dipende strettamente dalla natura delle applicazioni in esecuzione sulla VM. Tuttavia, in prima approssimazione, si puó considerare la quantitá di pagine modificate proporzionale al tempo di esecuzione stesso.
Assumiamo, inoltre, che tutte le VM considerate abbiano la stessa dimensione della memoria. Sappiamo che ció non è del tutto vero nelle *virtualized edge network*, e che il profilo di memoria di ogni macchina dipende dalla funzione di rete che deve svolgere, ció nonostante si accetta, momentaneamente, tale assunzione.
Infine, decidiamo di considerare costante il *rate di trasmissione* della memoria.

Tutte queste assunzioni permettono di semplificare le equazioni matematiche e di concentrarsi sulle prestazioni a livello macroscopico della migrazione di macchine virtuali all'interno di un Data Center, che poi è l'obiettivo principale di questa trattazione.

La modellizzazione delle prestazioni di una migrazione coinvolge, come accennato, diversi fattori: la dimensione della memoria della VM, le caratteristiche del workload [30], il rate di trasmissione della rete, e l'algoritmo di migrazione. Facciamo riferimento a [13] in cui si considera un approccio *pre-copy* che avviene in n iterazioni.
Sia V_i il volume dei dati trasmessi ad ogni iterazione, e T_i il tempo trascorso per ogni turno. Sia V_{mem} equivalente alla dimensione della memoria della VM, e T_0 rappresenti il tempo impiegato per trasferire l'istantanea della memoria. In Tabella 7.1 sono definiti altri importanti parametri.

V_{mig}	traffico di rete totale durante la migrazione
T_{mig}	tempo totale di migrazione
T_{down}	tempo durante il quale si ha interruzione del servizio
R	rate di trasmissione della memoria
D	rate con cui si modificano le pagine di memoria
V_{th}	valore di soglia per le pagine modificate:appena il volume dei dati trasmessi scende al di sotto di tale valore inizia la fase di stop-and-copy
W	WritableWorkingSet

Tabella 7.1: Alcuni parametri d' interesse.

I dati trasmessi all'i-esimo turno possono essere calcolati come:

$$V_i = \begin{cases} V_{mem} & \text{se } i = 0 \\ D \cdot T_{i-1} & \text{altrimenti} \end{cases} \tag{7.1}$$

Il tempo trascorso al turno iniziale risulta:

$$T_0 = \frac{V_{mem}}{R} \tag{7.2}$$

e il tempo trascorso all'i-esimo turno puó essere calcolato come proposto in eq. 7.3:

$$T_i = \frac{D \cdot T_{i-1}}{R} = \frac{V_{mem} \cdot D^i}{R^{i+1}} \tag{7.3}$$

Consideriamo la *dirtying-rate* sempre minore del *rate* di trasmissione delle pagine di memoria (in caso contrario, non avrebbe senso impiegare l'approccio *pre-copy*). Indichiamo con λ il rapporto tra D e R:

$$\lambda = \frac{D}{R} \tag{7.4}$$

Combinando (7.1), (7.3), (7.4), abbiamo in traffico di rete durante l'iterazione i-esima:

$$V_i = D \cdot \frac{V_{mem}}{R} \cdot \lambda^{i-1} = V_{mem} \cdot \lambda^i \tag{7.5}$$

Quindi, il traffico totale durante la migrazione puó essere scritto come:

$$V_{mig} = \sum_{i=0}^{n} V_i = V_{mem} \cdot \frac{1 - \lambda^{n+1}}{1 - \lambda} \tag{7.6}$$

Combinando (7.2), (7.3) si ottiene il tempo totale di migrazione:

$$T_{mig} = \sum_{i=0}^{n} T_i = \frac{V_{mem}}{R} \cdot \frac{1 - \lambda^{n+1}}{1 - \lambda} \tag{7.7}$$

Siccome T_{mig} è la durata della migrazione ed ha effetti negativi sulle prestazioni delle applicazioni, è un parametro chiave per stabilire se è conveniente migrare la VM.
Analizziamo ora il *downtime*. É composto da due parti: il tempo impiegato per trasferire le rimanenti pagine modificate durante la fase di stop-and-copy, T_n, e il tempo necessario per riavviare la VM sul nuovo host, chiamato T_{resume}. Quest'ultimo è poco variabile e puó essere rappresentato come una costante. Si puó scrivere il downtime:

$$T_{down} = T_n + T_{resume}. \tag{7.8}$$

Per una data VM, la dimensione della memoria, V_{mem}, e il valore di soglia delle pagine che possono essere trasferite durante la fase di stop-and-copy, V_{th}, sono fissate. Di conseguenza, la pre-copy iterativa convergerá tanto piú velocemente quanto piú λ sará piccola.
Ormai sappiamo che ogni OS esibisce un insieme di pagine *hot* che viene aggiornato molto frequentemente, chiamato Writable Working Set (WWS), che tali pagine si modificano con la stessa velocitá con cui il demone incaricato della migrazione riesce a trasferirle, e che quindi devono essere scartate durante le iterazioni di pre-copy. Per la maggior parte degli OS si osserva che le dimensioni dei WWS sono approssimativamente proporzionali alle pagine modificate in ogni turno di pre-copy:

$$W_i = \gamma \cdot V_i \tag{7.9}$$

dove γ dipende dalla *dirtying-rate* e dalla durata di ogni iterazione:

$$\gamma = -0,0463T_{i-1} - 0,0001D + 0,3586. \tag{7.10}$$

L'espressione (7.10) è ricavata empiricamente da un benchmark [30] che consiste in un'ampia varietá di applicazioni Java.
Dall'analisi sopra riportata, in [13] si conclude che una VM con un' "istantanea" della memoria e λ piccoli puó generare poco traffico e presentare un tempo totale di migrazione contenuto. Di conseguenza, tale Virtual Machine, puó essere la candidata ideale per la migrazione live.

7.2 Approccio simulativo

Per descrivere il sistema in esame e valutarne le prestazioni si puó impiegare un *simulatore orientato ad eventi*. Gli "eventi" sono i fenomeni caratteristici del sistema che ne determinano l'evoluzione.
In questa circostanza, sono "eventi" l'inizio di una migrazione, le iterazioni della pre-copy, e la fine della migrazione.
Il simulatore proposto (Appendice A) fará riferimento ai parametri di Tabella 7.2:

V	traffico trasmesso durante la migrazione
T_{mig}	tempo totale di migrazione
T_{dw}	tempo durante il quale si ha interruzione del servizio
R	rate di trasmissione della memoria
D	rate con cui si modificano le pagine di memoria
V_{th}	valore di soglia per le pagine modificate:appena il volume dei dati trasmessi scende al di sotto di tale valore inzia la fase di stop-and-copy

Tabella 7.2: Importanti parametri.

e accetta in ingresso almeno cinque valori, rispettivamente per:

- $seed$, che indica il "seme";

- $N - samples$, che indica il numero di eventi che s'intende valutare;

- $N - classes$, che indica il numero di classi che si vogliono considerare;

- $rho - class$, che indica la percentuale di tempo in cui il servitore è impegnato a servire la coda;

- $serv - class$, che indica per quanto tempo l'utente occupa mediamente il servitore.

A questo punto, si propone una valutazione quantitativa dei parametri d'interesse: si studierá il *tempo medio di migrazione* e il *downtime* in funzione della dimensione della memoria V, della soglia V_{th}, della *dirtying-rate* D, e del rate di trasmissione R.

7.3 Valutazioni quantitative dei parametri d'interesse

S'intende cominciare la valutazione studiando l'andamento del *tempo medio di migrazione*, $Tmig$, e del *downtime*, Tdw, in funzione della soglia V_{th}.
S'inseriranno nel simulatore i valori indicati di seguito:

- $R = 1$

- $D = 0.5$

- $V = 10$

- $seed = 1$

- $N - samples = 10$

- $N - classes = 1$

- $rho - class = 0.4$

- $serv - class = 1$

Gli andamenti sono riportati in Figura 7.1 e in Figura 7.2:

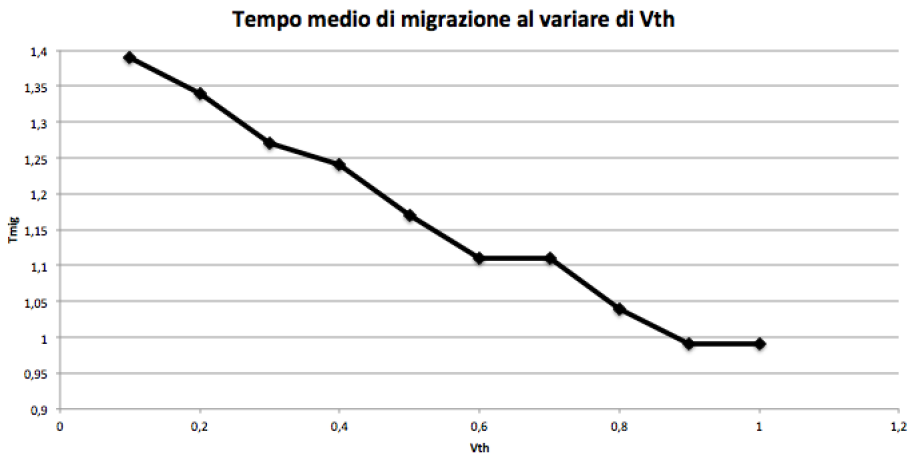


Figura 7.1: Tempo medio di migrazione in funzione di V_{th}.

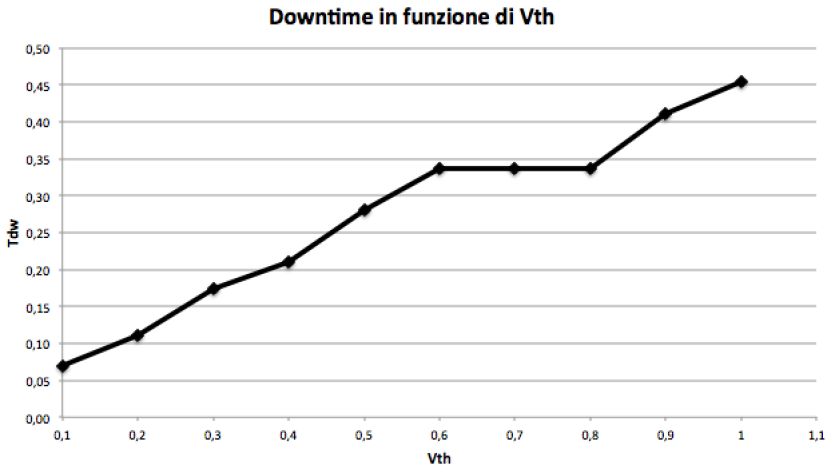


Figura 7.2: Downtime medio in funzione di V_{th}.

Come mostra Figura 7.1, all'aumentare dei KB di dati che possono essere trasmessi durante la fase di *stop-and-copy*, diminuisce il *tempo medio di migrazione*. Questo andamento rispecchia le aspettative: infatti, la durata della migrazione di una VM sará contenuta se durante la *stop-and-copy* possiamo trasmettere parte consistente della memoria.
Il *downtime* invece, all'aumentare di V_{th}, aumenta. Anche questo andamento

rispecchia quanto si attendeva: essendo il *downtime* definito come il tempo impiegato per trasferire i rimanenti KB della *stop-and-copy* (7.1.1), piú questa fase è duratura per dei numerosi dati da trasmettere, piú è grande il *downtime*.

Si vuole valutare, ora, l'andamento del *tempo medio di migrazione*, $Tmig$, e del *downtime*, Tdw, in funzione della dimensione della memoria V.
S'inseriranno nel simulatore i seguenti valori :

- $R = 1$
- $D = 0.5$
- $V_{th} = 0.1$
- $seed = 1$
- $N - samples = 10$
- $N - classes = 1$
- $rho - class = 0.4$
- $serv - class = 1$

Sono riportati in Figura 7.3 e in Figura 7.4 i due andamenti:

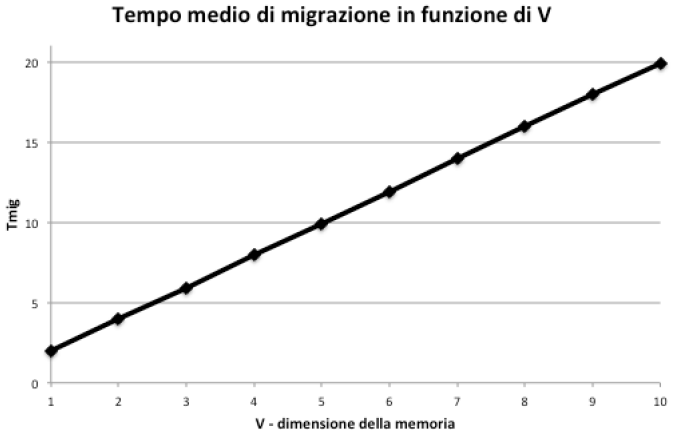


Figura 7.3: Tempo medio di migrazione in funzione di V.

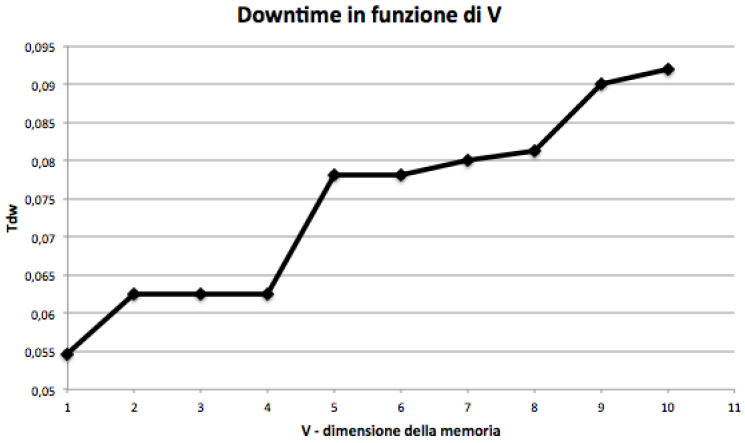


Figura 7.4: Downtime medio in funzione di V.

All'aumentare della dimensione della memoria che s'intende trasferire aumentano sia il *downtime*, sia il *tempo di migrazione*.
Figura 7.3 mostra come all'aumento di un unitá di V corrisponda una crescita doppia di $Tmig$.
La crescita di Tdw, invece, non risulta altrettanto lineare, come si puó notare osservando Figura 7.4.

Come precedentemente annunciato, si vuole determinare anche l'andamento del *tempo medio di migrazione* e del *downtime* in funzione della *dirtying-rate* D. Si fissano:

- $R = 1$

- $V = 10$

- $V_{th} = 0.1$

- $seed = 1$

- $N - samples = 10$

- $N - classes = 1$

- $rho - class = 0.4$

- $serv - class = 1$

Gli andamenti sono proposti in Figura 7.5 e in Figura 7.6, dove si puó notare anche come la *dirtying-rate* sia sempre mantenuta inferiore ad R:

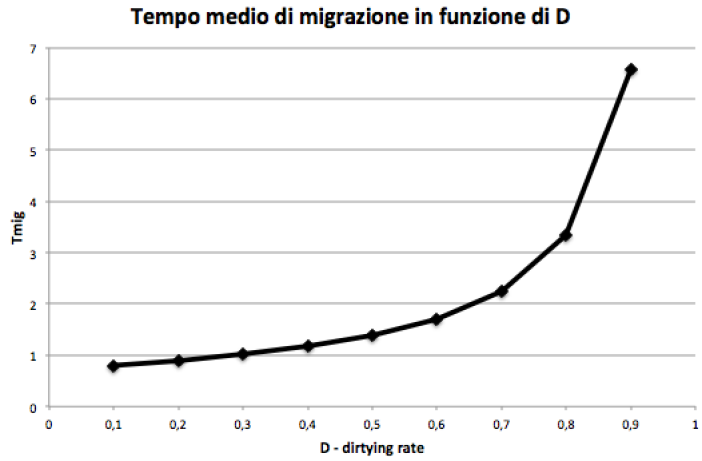


Figura 7.5: Tempo medio di migrazione in funzione di D.

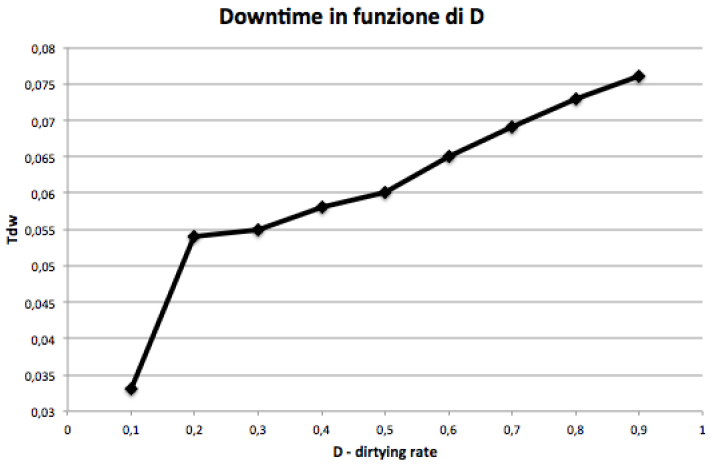


Figura 7.6: Downtime medio in funzione di D.

L'andamento del *tempo medio di migrazione* è crescente, non linearmente, con

D. In effetti, maggiore è il tasso con cui si modificano le pagine della memoria, maggiore sará il tempo da impiegare per avere copia coerente della memoria sul secondo sito.
Anche il *downtime* aumenta all'aumentare della *dirtying-rate*, seppur in maniera piú modesta.

Per concludere, si vuole valutare l'andamento del *tempo medio di migrazione* e del *downtime* in funzione del rate di trasmissione R. Si impone:

- $D = 0.5$

- $V = 10$

- $V_{th} = 0.1$

- $seed = 1$

- $N - samples = 10$

- $N - classes = 1$

- $rho - class = 0.4$

- $serv - class = 1$

In Figura 7.7 è presentato l'andamento del *tempo di migrazione* in funzione di R, mentre in Figura 7.8 l'andamento del *downtime*:

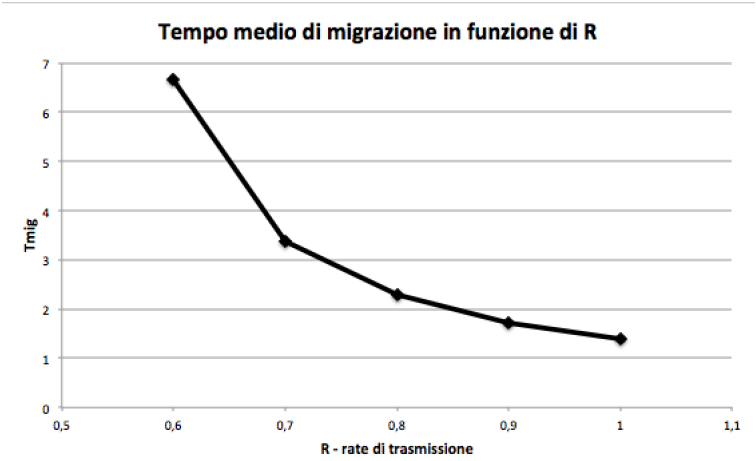


Figura 7.7: Tempo medio di migrazione in funzione di R.

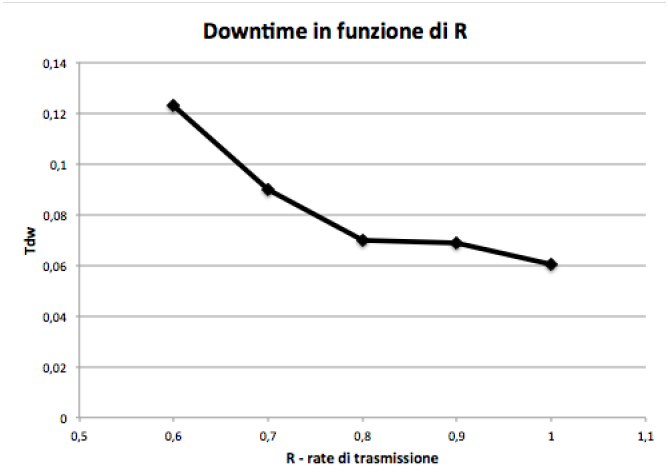


Figura 7.8: Downtime medio in funzione di R.

I grafici sopra riportati mostrano come $Tmig$ e Tdw diminuiscano all'aumentare di R: infatti, al crescere del tasso di trasmissione delle pagine della memoria, si riduce sia il tempo che s'impiega nella migrazione totale, sia il tempo impiegato per la fase di *stop-and-copy*.

7.4 Confronto tra l'approccio analitico e l'approccio simulativo

Per determinare la bontá degli approcci proposti nei paragrafi 7.1 e 7.2, si confrontano i risultati che forniscono al variare di V_{th}.

Fissando, per esempio:

- $V = 1$;
- $V_{th} = 0.1$;
- $\lambda = D/R = 0.5$;

adottando il *modello analitico base* si ottiene:

$V_1 = V = 1$	$T_1 = \frac{V_1}{R} = 1$
$V_2 = \frac{D}{R} * V_1 = 0.5$	$T_2 = \frac{V_2}{R} = 0.5$
$V_3 = \frac{D}{R} * V_2 = 0.25$	$T_3 = \frac{V_3}{R} = 0.25$
$V_4 = \frac{D}{R} * V_3 = 0.125$	$T_4 = \frac{V_4}{R} = 0.125$
$V_5 = \frac{D}{R} * V_4 = 0.0625$	$T_5 = \frac{V_5}{R} = 0.0625$
$V_5 < V_{th}$	iterazioni totali = 5;
	tempo totale di migrazione = $\sum_1^5 T_i$ = 1.9375;
	downtime = $T_5 = 0.0625$.

Tabella 7.3: Risultati - modello analitico.

mentre con l'*approccio simulativo*, impostando nel simulatore $seed = 1$, $N - samples = 1$, $N - classes = 1$, $rho - class = 0.8$ e $serv - class = 1$, si ottiene:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                1.937500                0.062500                5.000000

# x 5

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                1.937500                0.062500                5.000000
```

Figura 7.9: Approccio simulativo con V=1 e Vth=0.1.

Dal confronto tra i due approcci si puó notare come i valori del numero d'iterazioni, del tempo di migrazione e del downtime coincidono.

Fissando invece:

- $V = 1$;
- $V_{th} = 0.3$;
- $\lambda = \frac{D}{R} = 0.5$;

con il *modello analitico base* si ottiengono i risultati proposti in Tabella 7.4:

$V_1 = V = 1$	$T_1 = \frac{V_1}{R} = 1$
$V_2 = \frac{D}{R} * V_1 = 0.5$	$T_2 = \frac{V_2}{R} = 0.5$
$V_3 = \frac{D}{R} * V_2 = 0.25$	$T_3 = \frac{V_3}{R} = 0.25$
$V_3 < V_{th}$	"stop-and-copy
	iterazioni totali = 3;
	tempo totale di migrazione = $\sum_1^3 T_i$ = 1.75;
	downtime = $T_3 = 0.25$.

Tabella 7.4: Risultati con $V_{th} = 0.3$ - modello analitico.

mentre con l' *approccio simulativo*, attribuendo a seed, a $N - samples$, a $N - classes$ e a $serv - class$ il valore 1 e a $rho - class$ il valore 0.8, si ottengono i risultati proposti in Figura 7.10:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                1.750000                0.250000                3.000000

# x 3

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                1.750000                0.250000                3.000000
```

Figura 7.10: Approccio simulativo con V=1 e Vth=0.3.

Come si puó notare, anche questa volta i risultati coincidono.

Fissando ora:

- $V = 1$;
- $V_{th} = 1.1$;
- $\lambda = \frac{D}{R} = 0.5$;

con il *modello analitico base* otteniamo:

$V_1 = V = 1$	$T_1 = \frac{V_1}{R} = 1$
$V_1 < V_{th}$	iterazioni totali = 1;
	tempo totale di migrazione = $T_1 = 1$;
	downtime = $T_1 = 1$.

Tabella 7.5: Risultati con $V_{th} = 1.1$ - modello analitico

mentre con l'*approccio simulativo*, mantenendo seed, $N - samples$, $N - classes$ e $serv - class$ pari a 1, e $rho - class$ pari a 0.8, si ottiengono i risultati riportati in Figura 7.11.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                1.000000                1.000000                1.000000

# x 1

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                1.000000                1.000000                1.000000
```

Figura 7.11: Approccio simulativo con V=1 e Vth=1.1.

Ancora una volta quest'ultimi coincidono.

Si puó concludere che il *simulatore ad eventi* proposto descrive il sistema, e porta agli stessi risultati, del modello matematico.

Fissiamo allora il valore di V_{th} a 0.1 e verifichiamo, attraverso il simulatore, i valori degli indici di prestazione della migrazione al variare di $N - samples$, quindi di $rho - class$.
In Figura 7.12 sono indicate le dimensioni della memoria, ottenute in maniera casuale, considerando seed = 0, $N-samples = 5$, $N-classes = 1$, $rho-class =$

0.8 e $serv-class = 1$, e sono evidenziati i valori del tempo medio di migrazione, del numero medio d'iterazioni, e del downtime medio.
Di seguito è proposto il codice impiegato per generare la dimensione della memoria:

```
double expon(double param) {

 // Declare some useful variables
  int rnd_num;
 double unif,val;

 // Generate a uniform random number between 0 and 1
 rnd_num = rand();
 if (rnd_num == RAND_MAX) rnd_num--;
 unif = (double)rnd_num/RAND_MAX;

 // Transform the uniform random variable into an
       exponential one using the inverse function rule
 val = -log(1.0-unif)/param;
 return val;
}

double serv(char pclass) {

 // Returns an exponential service time
 return expon(mu[pclass]);
}

void initialize() {
  int j;
  // Initialize all the global variables
and counters

  k = 0; tfree = 0.0; now = 0.0;
  tot_arrivals = 0; tot_departures = 0;
  tot_w = 0.0; tot_mig = 0.0;
  tot_downtime = 0.0;
```

```
  for (j = 0; j < C; j++) {
    arrivals[j] = 0; w[j] = 0.0;Tdw[j] = 0.0; Tmig[j] = 0.0;
    insert_new_event(expon(lambda[j]),MIGRATION_REQ,j,serv(j), 0,0,0,0);
    q[j]=NULL;
  }
}
```

```
# Memory Size 0.652744

# Memory Size 0.913382

# Memory Size 0.521232

# Memory Size 0.872272

# Memory Size 2.297851

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             1.002984                2.036961                0.066032                4.600000

# x 20

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                1.002984                2.036961                0.066032                4.600000
```

Figura 7.12: Approccio simulativo con V variabile e Vth=0.1.

Ripetendo la simulazione, impostando $N-samples = 10$, otteniamo quanto riportato in Figura 7.13.

```
# Memory Size 0.054807

# Memory Size 2.466730

# Memory Size 2.221626

# Memory Size 0.913617

# Memory Size 0.106970

# Memory Size 0.534183

# Memory Size 1.042391

# Memory Size 3.697120

# Memory Size 2.624065

# Memory Size 3.067998

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             8.599133                3.277954                0.067947                4.800000

# x 39

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                8.599133                3.277954                0.067947                4.800000
```

Figura 7.13: Approccio simulativo con V variabile e Vth=0.1.

Continuando a mantenere V_{th}, valore di soglia per il traffico, fissato a 0.1, e ad ottenere in maniera casuale il valore della variabile V che rappresenta il traffico totale di migrazione, verifichiamo i tempi medi di downtime, di migrazione e il numero medio di iterazioni di *pre-copy* nel caso in cui il numero di eventi che si intende considerare sia elevato: $N - samples = 100000$.
I valori degli altri parametri d'ingresso al simulatore rimangono: seed = 0, $N - classes = 1$, $rho - class = 0.8$, $serv - class = 1$:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             3334.350141             1.924929                0.069477                4.085100

# x 30852

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                3334.350141             1.924929                0.069477                4.085100
```

Figura 7.14: Approccio simulativo con V variabile, Vth=0.1 e $N - samples$=100000.

Com' è possibile osservare dai risultati dell'ultima simulazione proposti in Figura 7.14, il tempo di attesa medio è inaccettabile poiché intollerabilmente elevato. Questo perché si è attribuito al parametro $rho - class$, che indica che la percentuale di tempo in cui il servitore è occupato a servire una richiesta, il valore 0.8, che è piuttosto elevato e, se si considerano numerosi eventi, porta al rapido crearsi di una coda che il servitore sará lento a smaltire.
Per ottenere il carico effettivo del sistema s'introduce la nuova variabile "load", definita come prodotto tra il tempo medio di migrazione e lambda:

```
double load[MAX_CLASSES]; // carico effettivo del sistema

load[j] = Tmig[j]* lambda[j];

printf(" n# Carico effettivo \%f  n",load[j]);
```

Ripetendo la simulazione per $N - samples = 100000$, il carico effettivo del sistema è riportato in Figura 7.15.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             33988.771214            1.922079                0.069629                4.081120

# Carico effettivo 1.537663

# x 308113

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                33988.771214            1.922079                0.069629                4.081120
```

Figura 7.15: Carico Effettivo con $N - samples$=100000 e $rho - classes$=0.8.

Riducendo il valore di $rho - class$ a 0.4, per esempio, si ottengono dei tempi di attesa tollerabili anche considerando un numero notevole di eventi.
Fissiamo seed = 0, $N - samples = 1000000$, $N - classes = 1$, $rho - class = 0.4$ e $serv - class = 1$.
Dalla simulazione ricaviamo i risultati di Figura 7.16.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             6.749120                1.931694                0.069668                4.088188

# Carico effettivo 0.772678

# x 3088189

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                6.749120                1.931694                0.069668                4.088188
```

Figura 7.16: Tempo di attesa medio ridotto con $rho - class = 0.4$.

Capitolo 8

Migrazione di insiemi di Virtual Machine

In [23] è descritta la migrazione di un Data Center attraverso le WAN: sono indicati i requisiti di rete, i requisiti per la replicazione dei dati, ecc. ma non è specificato come avviene la migrazione di piú macchine virtuali (si fa riferimento sempre ad un solo virtual server).
Puó, dunque, essere interessante capire da quale VM iniziare la migrazione, quali caratteristiche deve avere, quali parametri monitorare.
L'obiettivo di [13] è proprio quello di cercare di capire quale VM, tra tutte quelle presenti in un Data Center, conviene migrare per prima: nell'articolo si suggerisce di prestare particolare attenzione alla dimensione della memoria, al *rate* di trasmissione della pagine e di scegliere con cura l'algoritmo da adottare per tale trasmissione.
Focalizzando l'attenzione sulle pagine di memoria delle VM, ipotizzando il rate di trasmissione di tali pagine costante e considerando omogenea la dimensione della memoria per tutte le macchine, si propone la simulazione di migrazione di M macchine virtuali prima *in serie*, poi *in parallelo*. A tal proposito, si suppone che la migrazione del set di macchine virtuali non puó considerarsi conclusa finché non sono migrate tutte le VM da un sito all'altro.
Si propone, inoltre, il confronto dei risultati ottenuti.

8.1 Migrazione di macchine virtuali in serie

Consideriamo M macchine virtuali identiche tra loro, e supponiamo di volerle migrare *in serie*, cioé la migrazione di una VM puó iniziare solo al termine della migrazione della macchina precedente.

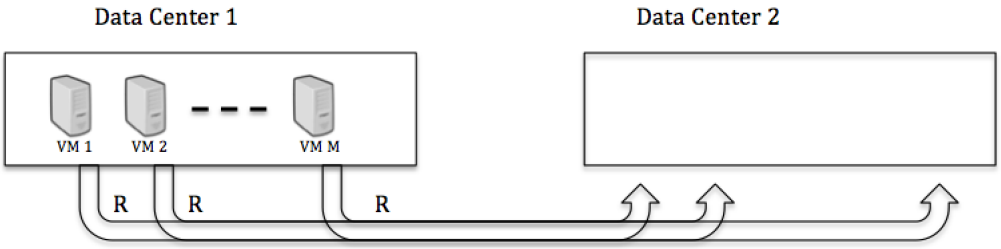


Figura 8.1: Migrazione in serie.

Mediante l'utilizzo del simulatore precedentemente proposto, ma corretto opportunamente (Appendice B), si vogliono determinare il *tempo medio di migrazione* e il *downtime*.

8.1.1 Confronto tra l'approccio analitico e l'approccio simulativo.

Per coerenza con la trattazione analitica di paragrafo 7.1.1 si riscrive il tempo di *downtime* come in eq. 8.1:

$$T_{dw}^{(serie)} = \frac{V}{R}\lambda^{n(serie)} + (M-1)\frac{V}{R}\frac{1-\lambda^{n(serie)+1}}{1-\lambda} + T_{resume} \tag{8.1}$$

dove $n(serie)$ è il numero di iterazioni del processo di migrazione *in serie*, definito in eq. 8.2:

$$n(serie) = \left\lceil \log_{\lambda} \frac{V_{\text{th}}}{V} \right\rceil \tag{8.2}$$

Il *tempo medio di migrazione*, sempre con riferimento alla trattazione analitica proposta in 7.1.1, si puó scrivere come:

$$T_{mig}^{(serie)} = M \frac{V}{R} \frac{1 - \lambda^{n^{(serie)}+1}}{1 - \lambda} \tag{8.3}$$

Per verificare l'affidabilitá del simulatore per la migrazione delle VM in serie, confrontiamo i risultati ottenuti con l'approccio analitico e con quello simulativo.

Impostiamo, per semplicitá di calcolo:

- $R = 1$;
- $D = 0.5$;
- $V = 2$;
- $M = 2$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.1$;
- $serv - class = 1$;

e facciamo variare V_{th}.
Per la prima simulazione poniamo $V_{th} = 0.1$.
Con l'approccio analitico si ottengono i valori di Tabella 8.1:

$n(serie) = \lceil \log_{0.5} 0.05 \rceil = \lceil 4.32 \rceil = 5$
$T_{dw}^{(serie)} = 2 \cdot 0.5^5 + 2 \frac{1-0.5^6}{1-0.5} = 4$
$T_{mig}^{(serie)} = 2 \cdot 2 \frac{1-0.5^6}{1-0.5} = 7.875$

Tabella 8.1: Indici di prestazione ottenuti con il modello analitico - Migrazione in serie

mentre con l'approccio simulativo si è ottenuto:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             12.929376               7.875000                4.000000                5.000000

# Carico effettivo 0.787500

# Nset 1

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                12.929376               7.875000                4.000000                5.000000
```

Figura 8.2: Indici di prestazione ottenuti con il simulatore - Migrazione in serie.

Com' è possibile notare comparando Tabella 8.1 e Figura 8.2 i risultati coincidono.

Poniamo ora $V_{th} = 0.4$.
Con l'approccio analitico si ottengono i valori di Tabella 8.2:

$n(serie) = \lceil \log_{0.5} 0.2 \rceil = \lceil 2.32 \rceil = 3$
$T_{dw}^{serie} = 2 \cdot 0.5^3 + 2\,\frac{1-0.5^4}{1-0.5} = 4$
$T_{mig}^{(serie)} = 2 \cdot 2\,\frac{1-0.5^4}{1-0.5} = 7.5$

Tabella 8.2: Indici di prestazione ottenuti con il modello analitico - Migrazione in serie

I valori ottenuti con l'approccio simulativo sono invece proposti in Figura 8.3:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             12.179376               7.500000                4.000000                3.000000

# Carico effettivo 0.750000

# Nset 1

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                12.179376               7.500000                4.000000                3.000000
```

Figura 8.3: Indici di prestazione ottenuti con il simulatore - Migrazione in serie.

Anche per $V_{th} = 0.4$ i risultati di Tabella 8.2 e Figura 8.3 coincidono.

Decidiamo allora di fissare V_{th} a 0.1 e di variare la dimensione delle memorie delle VM.
Si pone $V = 4$, mentre gli altri parametri rimangono invariati:

- $R = 1$;
- $D = 0.5$;
- $V_{th} = 0.1$;
- $M = 2$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.1$;
- $serv - class = 1$;

Con l'approccio analitico si ottengono i valori di Tabella 8.3:

$n(serie) = \lceil \log_{0.5} 0.025 \rceil = \lceil 5.32 \rceil = 6$
$T_{dw}^{serie} = 4 \cdot 0.5^6 + 4 \frac{1-0.5^7}{1-0.5} = 8$
$T_{mig}^{(serie)} = 2 \cdot 4 \frac{1-0.5^7}{1-0.5} = 15.875$

Tabella 8.3: Indici di prestazione ottenuti con il modello analitico - Migrazione in serie

mentre con l'approccio simulativo si ottiene quanto presentato in Figura 8.4:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             28.929376               15.875000               8.000000                6.000000

# Carico effettivo 1.587500

# Nset 1

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                28.929376               15.875000               8.000000                6.000000
```

Figura 8.4: Indici di prestazione ottenuti con il simulatore - Migrazione in serie.

Ancora una volta i valori degli indici di prestazione d'interesse coincidono.

Si decide, prima di confermare la validitá del simulatore, di variare la *dirtying-rate* mantenendo fissi gli altri parametri:

- $R = 1$;
- $V = 2$;
- $V_{th} = 0.1$;
- $M = 2$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.1$;
- $serv - class = 1$;

Poniamo $D = 0.2$, cosí che il prodotto $M \cdot D$ rimanga inferiore ad R.
I risultati ottenuti impiegando l'approccio analitico e l'approccio simulativo sono presentati in Tabella 8.4 e in Figura 8.5, rispettivamente.

$n(serie) = \lceil \log_{0.2} 0.05 \rceil = \lceil 1.86 \rceil = 2$
$T_{dw}^{serie} = 2 \cdot 0.2^2 + 2\,\frac{1-0.2^3}{1-0.2} = 2.56$
$T_{mig}^{(serie)} = 2 \cdot 2\,\frac{1-0.2^3}{1-0.2} = 4.96$

Tabella 8.4: Indici di prestazione ottenuti con il modello analitico - Migrazione in serie

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             7.099376                4.960000                2.560000                2.000000

# Carico effettivo 0.496000

# Nset 1

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                7.099376                4.960000                2.560000                2.000000
```

Figura 8.5: Indici di prestazione ottenuti con il simulatore - Migrazione in serie.

Il *tempo medio di migrazione* e il *downtime* sono gli stessi nei due casi.

Ne consegue la validitá del *simulatore di migrazione di M macchine virtuali in serie* proposto.

8.1.2 Simulazioni di migrazioni *in serie* di reti di macchine virtuali

A questo punto ipotizziamo, per semplicitá, M = 4 e fissiamo seed = 1, $N - samples = 8$ $N - classes = 1$, $rho - class = 0.4$ e $serv - class = 1$. I risultati della simulazione sono i seguenti:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             12.612340               5.434924                1.092191        3.375000

# Carico effettivo 2.173969

# Nset 2

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                12.612340               5.434924                1.092191        3.375000
```

Figura 8.6: Indici di prestazione - Migrazione in serie.

Come è possibile notare da Figura 8.6, il *carico effettivo* è maggiore di 1. Modifichiamo quindi il valore del parametro $rho - class$ (rho-class = 0.1) affinché ció non sia piú vero, e riportiamo i risultati del simulatore in Figura 8.7.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             6.248625                5.434924                1.092191                3.375000

# Carico effettivo 0.543492

# Nset 2

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                6.248625                5.434924                1.092191                3.375000
```

Figura 8.7: Indici di prestazione con rho-class = 0.1 - Migrazione in serie.

Aumentando il numero degli eventi considerati, per esempio $N - samples = 10000$, a paritá di valore degli altri parametri, si ottiene quanto riportato in Figura 8.8 :

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             11.670114               7.730383                1.464263                4.083600

# Carico effettivo 0.773038

# Nset 2500

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                11.670114               7.730383                1.464263                4.083600
```

Figura 8.8: Indici di prestazione con $N - samples$ elevato - Migrazione in serie.

Se aumentiamo il numero di macchine virtuali di un set, ad esempio poniamo M = 10, otteniamo:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             29.175285               19.325958               1.750253                4.083600

# Carico effettivo 1.932596

# Nset 1000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                29.175285               19.325958               1.750253                4.083600
```

Figura 8.9: Indici di prestazione con M=10 - Migrazione in serie.

Essendo il *carico effettivo* nuovamente maggiore di 1, riduciamo ulteriormente $rho-class$ ponendolo uguale a 0.04. Di seguito si propongono i risultati.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             22.544394               19.325958               1.750253                4.083600

# Carico effettivo 0.773038

# Nset 1000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                22.544394               19.325958               1.750253                4.083600
```

Figura 8.10: Indici di prestazione con M=10 e $rho-class$ = 0,04 - Migrazione in serie.

8.2 Migrazione di macchine virtuali in parallelo

Consideriamo ancora M macchine virtuali identiche tra loro, e supponiamo di volerle migrare *in parallelo*: ció vuol dire che la banda disponibile dovrá essere condivisa tra gli M elementi.
Come conseguenza, ogni VM avrá il rate di trasmissione delle pagine ridotto a $\frac{R}{M}$.

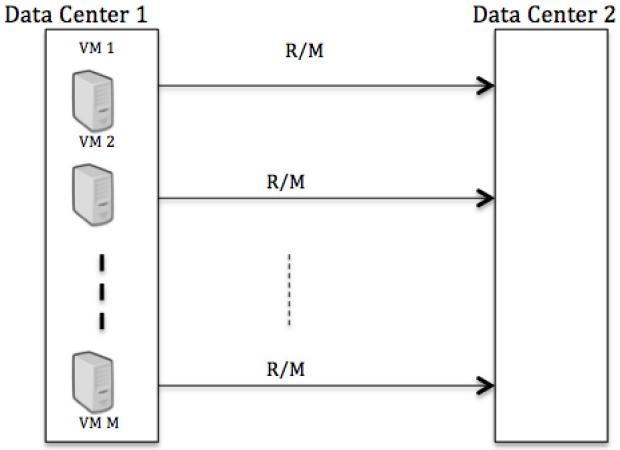


Figura 8.11: Migrazione in parallelo.

8.2.1 Confronto tra l'approccio analitico e l'approccio simulativo.

Per continuare a mantenere la coerenza con la trattazione analitica di paragrafo 7.1.1 si riscrive il tempo di *downtime* come proposto in eq. 8.4:

$$T_{dw}^{(parallelo)} = M \cdot \frac{V}{R} \cdot (M\lambda)^{n(parallelo)} + T_{resume} \tag{8.4}$$

dove $n(parallelo)$ è il numero di iterazioni del processo di migrazione *in parallelo*, definito in eq. 8.5:

$$n(parallelo) = \left\lceil \log_{(M\,\lambda)} \frac{V_{\text{th}}}{V} \right\rceil \tag{8.5}$$

Per il *tempo medio di migrazione* si fará riferimento ad eq. 8.6:

$$T_{mig}^{(parallelo)} = M \cdot \frac{V}{R} \cdot \frac{1 - (M\,\lambda)^{n^{(parallelo)}+1}}{1 - (M\,\lambda)} \tag{8.6}$$

Si vuole controllare l'adeguatezza del simulatore per la migrazione delle VM in parallelo. A tal proposito, si confrontano i risultati ottenuti con l'approccio analitico e con quello simulativo.

Ipotizziamo, per semplicitá:

- $R = 1$;
- $D = 0.25$;
- $V = 2$;
- $M = 2$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.1$;
- $serv - class = 1$;

e facciamo variare V_{th}.
Poniamo, inizialmente, $V_{th} = 0.3$.
Con l'approccio analitico si ottengono i risultati proposti in Tabella 8.5:

$n(parallelo) = \lceil \log_{0.5} 0.15 \rceil = \lceil 2.73 \rceil = 3$
$T_{dw}^{parallelo} = 2 \cdot 2 \cdot (2 \cdot 0.25)^2 = 0.5$
$T_{mig}^{(parallelo)} = 2 \cdot 2 \frac{1-(2 \cdot 0.25)^4}{1-(2\,0.25)} = 7.5$

Tabella 8.5: Indici di prestazione ottenuti con il modello analitico - Migrazione in parallelo

Con l'approccio simulativo si ottiene:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                7.500000                0.500000                3.000000

# Carico effettivo 0.750000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                7.500000                0.500000                3.000000
```

Figura 8.12: Indici di prestazione ottenuti con il simulatore - Migrazione in parallelo.

I valori degli indici di prestazione presentati in Tabella 8.5 e in Figura 8.12 coincidono.

Poniamo, questa volta, $V_{th} = 0.1$.
Con l'approccio analitico si ottengono i risultati proposti in Tabella 8.6:

$n(parallelo) = \lceil \log_{0.5} 0.05 \rceil = \lceil 4.32 \rceil = 5$
$T_{dw}^{parallelo} = 2 \cdot 2 \cdot (2 \cdot 0.25)^5 = 0.125$
$T_{mig}^{(parallelo)} = 2 \cdot 2 \frac{1-(2 \cdot 0.25)^6}{1-(2\,0.25)} = 7.875$

Tabella 8.6: Indici di prestazione ottenuti con il modello analitico - Migrazione in parallelo

mentre il simulatore restituisce:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                7.875000                0.125000                5.000000

# Carico effettivo 0.787500

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                7.875000                0.125000                5.000000
```

Figura 8.13: Indici di prestazione ottenuti con il simulatore - Migrazione in parallelo.

Come si puó osservare da Tabella 8.6 e Figura 8.13 i risultati, ancora una volta, coincidono.

Fissiamo, allora, V_{th} a 0.1 e facciamo variare V.
Poniamo $V = 3$, mentre manteniamo invariati i parametri riproposti di seguito:

- $R = 1$;
- $D = 0.25$;
- $V_{th} = 0.1$;
- $M = 2$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.1$;
- $serv - class = 1$;

L'approccio analitico consente di arrivare ai seguenti risultati:

$n(parallelo) = \lceil \log_{0.5} 0.033 \rceil = \lceil 4.9 \rceil = 5$
$T_{dw}^{parallelo} = 2 \cdot 3 \cdot (2 \cdot 0.25)^5 = 0.1875$
$T_{mig}^{(parallelo)} = 2 \cdot 3 \frac{1-(2 \cdot 0.25)^6}{1-(2\,0.25)} = 11.8125$

Tabella 8.7: Indici di prestazione ottenuti con il modello analitico - Migrazione in parallelo

L'approccio simulativo porta a quelli presentati in Figura 8.14:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                11.812500               0.187500                5.000000

# Carico effettivo 1.181250

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                11.812500               0.187500                5.000000
```

Figura 8.14: Indici di prestazione ottenuti con il simulatore - Migrazione in parallelo.

Come si puó notare, in Tabella 8.7 e Figura 8.14 si hanno gli stessi valori per il *tempo medio di migrazione* e per il *downtime*.

Imponiamo $V = 5$. I risultati ottenuti analiticamente sono proposti di seguito:

$n(parallelo) = \lceil \log_{0.5} 0.02 \rceil = \lceil 5.64 \rceil = 6$
$T_{dw}^{parallelo} = 2 \cdot 5 \cdot (2 \cdot 0.25)^6 = 0.15625$
$T_{mig}^{(parallelo)} = 2 \cdot 5 \frac{1-(2 \cdot 0.25)^7}{1-(2\,0.25)} = 19.84375$

Tabella 8.8: Indici di prestazione ottenuti con il modello analitico - Migrazione in parallelo

mentre quelli ottenuti con il simulatore sono riportati in Figura 8.15:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                19.843750               0.156250                6.000000

# Carico effettivo 1.984375

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                19.843750               0.156250                6.000000
```

Figura 8.15: Indici di prestazione ottenuti con il simulatore - Migrazione in parallelo.

Decidiamo, a questo punto, d'imporre $D = 0.1$ e di mantenere costanti i valori dei parametri riportati di seguito:

- $R = 1$;
- $V = 2$;
- $V_{th} = 0.1$;
- $M = 2$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.1$;
- $serv - class = 1$;

L'approccio analitico porta ai seguenti risultati:

$n(parallelo) = \lceil \log_{0.2} 0.05 \rceil = \lceil 1.86 \rceil = 2$
$T_{dw}^{parallelo} = 2 \cdot 2 \cdot (2 \cdot 0.1)^2 = 0.16$
$T_{mig}^{(parallelo)} = 2 \cdot 2 \frac{1-(2 \cdot 0.1)^3}{1-(2\,0.1)} = 4.96$

Tabella 8.9: Indici di prestazione ottenuti con il modello analitico - Migrazione in parallelo

Attraverso il simulatore per la migrazione in parallelo si ottiene:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                4.960000                0.160000                2.000000

# Carico effettivo 0.496000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                4.960000                0.160000                2.000000
```

Figura 8.16: Indici di prestazione ottenuti con il simulatore - Migrazione in parallelo.

Ancora una volta i risultati sono gli stessi.

Sia, allora, $D = 0.3$.
Matematicamente otteniamo:

$n(parallelo) = \lceil \log_{0.6} 0.05 \rceil = \lceil 5.86 \rceil = 6$
$T_{dw}^{parallelo} = 2 \cdot 2 \cdot (2 \cdot 0.3)^6 = 0.186624$
$T_{mig}^{(parallelo)} = 2 \cdot 2 \frac{1-(2 \cdot 0.3)^7}{1-(2\,0.3)} = 9.72$

Tabella 8.10: Indici di prestazione ottenuti con il modello analitico - Migrazione in parallelo

mentre attraverso il simulatore per la migrazione in parallelo si ottiene:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                9.720064                0.186624                6.000000

# Carico effettivo 0.972006

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                9.720064                0.186624                6.000000
```

Figura 8.17: Indici di prestazione ottenuti con il simulatore - Migrazione in parallelo.

Alla luce dei risultati identici ottenuti con i due approcci, si puó validare il

simulatore per la *migrazione di M virtual machine in parallelo* proposto in Appendice C.

8.2.2 Simulazioni di migrazioni *in parallelo* di reti di macchine virtuali

Mediante l'utilizzo del simulatore per la migrazione in parallelo si vogliono determinare il *tempo medio di migrazione* e il *downtime*.

Ipotizziamo M = 4, fissiamo seed = 1, $N - samples = 8$, $N - classes = 1$, $rho - class = 0.4$ e $serv - class = 1$ e simuliamo. Ricordiamo che $M \cdot D < R$, per cui poniamo $D = 0.2$. I risultati sono riportati in Figura 8.18.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             25.181127               13.062601               0.279117                6.750000

# Carico effettivo 5.225040

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                25.181127               13.062601               0.279117                6.750000
```

Figura 8.18: Indici di prestazione - Migrazione in parallelo.

Consideriamo anche il caso in cui M = 10, quindi poniamo $D = 0.05$, seed = 1, $N-samples = 10000$, $N-classes = 1$, $serv-class = 1$, e $rho-class = 0.04$ cosí da tenere il *carico effettivo* inferiore a 1:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             61.717317               19.325958               0.696283                3.083600

# Carico effettivo 0.773038

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                61.717317               19.325958               0.696283                3.083600
```

Figura 8.19: Indici di prestazione con M e $N - samples$ elevati - Migrazione in parallelo.

A questo punto, per poter comparare gli indici di prestazione ottenuti dalle simulazioni delle migrazioni *in serie* e *in parallelo*, riponiamo seed = 1, M = 10, $N - samples = 10000$, $N - classes = 1$, $rho - class = 0.04$ e $serv - class = 1$ nel simulatore di migrazione *in serie* e riproponiamo i risultati:

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             22.544394               19.325958               1.750253                4.083600

# Carico effettivo 0.773038

# Nset 1000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                22.544394               19.325958               1.750253                4.083600
```

Figura 8.20: Indici di prestazione con $rho-class = 0.04$ - Migrazione in serie.

Dal confronto tra Figura 8.19 e Figura 8.20 si puó notare che, nell'ipotesi di migrazione *in serie*, il *downtime*, tempo durante il quale si verifica l'interruzione del servizio, è maggiore rispetto a quello calcolato nell'ipotesi di migrazione *in parallelo*.

Supponiamo di mantenere fissi i valori dei seguenti parametri:

- $R = 1$;
- $D = 0.05$;
- $V = 1$;
- $V_{th} = 0.1$;
- $M = 10$;
- $seed = 1$;
- $N - samples = 10000$;
- $N - classes = 1$;
- $rho - class = 0.04$;
- $serv - class = 1$;

e ripetiamo il confronto. I risultati sono mostrati nelle Figure 8.21 e 8.22.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             26.761999               19.375000               0.625000                4.000000

# Carico effettivo 0.775000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                26.761999               19.375000               0.625000                4.000000
```

Figura 8.21: Indici di prestazione con $D = 0.05$ Migrazione in parallelo.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             10.951313               10.500000               0.950000                2.000000

# Carico effettivo 0.420000

# Nset 1000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                10.951313               10.500000               0.950000                2.000000
```

Figura 8.22: Indici di prestazione con $D = 0.05$ - Migrazione in serie.

Si ribadisce, dunque, come il *downtime* sia minore nel caso di migrazione *in parallelo*.
Al contrario, il *tempo medio di migrazione* è minore se si decide di migrare le M virtual machine *in serie*.

Per approfondire la comparazione tra i due processi, si propongono quattro grafici in cui sono messi a confronto gli indici di prestazione della *migrazione in serie* e della *migrazione in parallelo* in funzione del numero di macchine virtuali in un set, e della *dirtying-rate*.
Nel primo caso si fissano:

- $R = 1$;
- $D = 0.2$;
- $V = 5$;
- $V_{th} = 0.1$;
- $seed = 1$;
- $N - samples = 1$;
- $N - classes = 1$;

- $rho - class = 0.1$;
- $serv - class = 1$;

mentre nel secondo s'impongono:

- $R = 1$;
- $M = 2$;
- $V = 5$;
- $V_{th} = 0.1$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.1$;
- $serv - class = 1$;

È fondamentale che, in entrambe le situazioni, si mantenga il prodotto tra M e D inferiore ad R: $M \cdot D < R$; in caso contrario, non sarebbe piú appropriata l'adozione dell'approccio *pre-copy*.

In Figura 8.23 e 8.24 sono riportati gli andamenti del *tempo di migrazione* e del *downtime*, rispettivamente, in funzione del numero di macchine in un set.

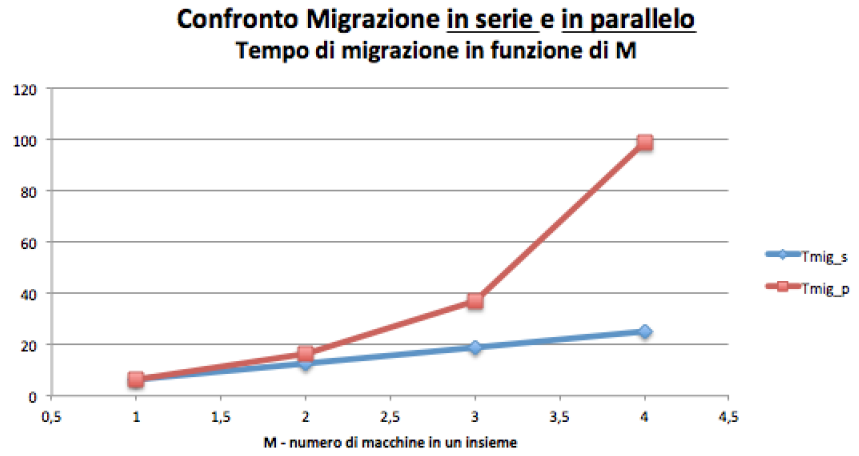


Figura 8.23: Confronto tra il tempo di migrazione della migrazione in serie e della migrazione in parallelo.

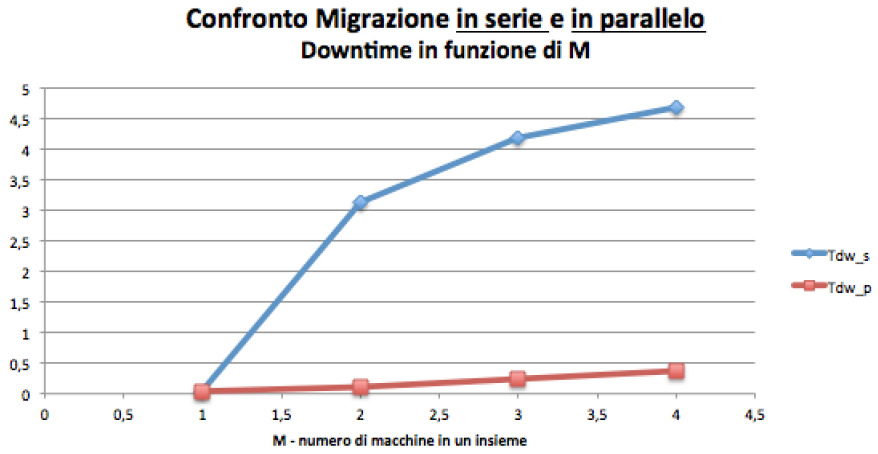


Figura 8.24: Confronto tra il downtime della migrazione in serie e della migrazione in parallelo.

In Figura 8.23 è evidente come il *tempo di migrazione* del processo *in parallelo* sia decisamente maggiore di quello del processo *in serie* al variare di M.

La Figura 8.24 mostra, invece, come il *downtime* del processo di migrazione *in serie* sia maggiore di quello del processo *in parallelo*. Tale considerazione era prevedibile ricordando le definizioni del tempo di *downtime*:

$$T_{dw}^{(serie)} = T'_{dw} + (M-1)\,T'_{mig} = T'_{dw}\,M + (T'_{mig} - T'_{dw})(M-1)$$

per il calcolo del *downtime* nella migrazione in serie,

$$T_{dw}^{(parallelo)} = T'_{dw} \cdot M$$

per il calcolo del *downtime* nella migrazione in parallelo.

Si propone una visione piú accurata del *downtime* della migrazione in parallelo in funzione del numero di macchine in un set in Figura 8.25:

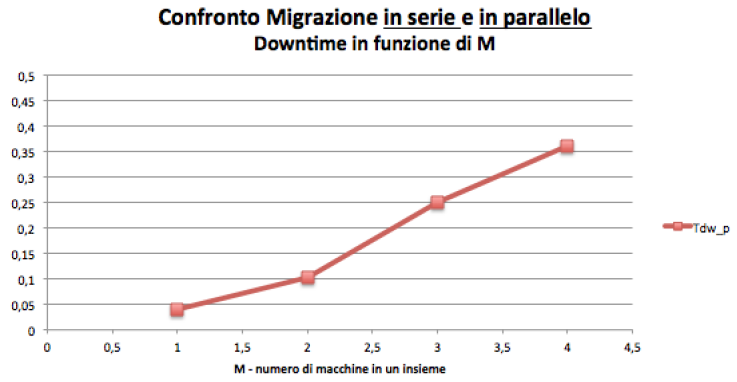


Figura 8.25: Confronto tra il downtime della migrazione in serie e della migrazione in parallelo: particolare della migrazione in parallelo.

In Figura 8.26 e 8.27 sono presentati gli andamenti del *tempo di migrazione* e del *downtime* in funzione della *dirtying-rate*:

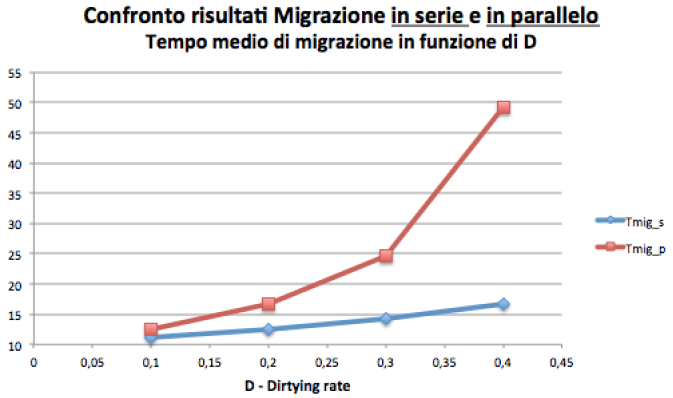


Figura 8.26: Confronto tra il tempo di migrazione della migrazione in serie e della migrazione in parallelo.

Nella Figura 8.26 si puó notare come i *tempi di migrazione* aumentino all'aumentare della *dirtying-rate*. Infatti, come discusso nel paragrafo 7.3, il tempo impiegato per trasferire una copia coerente della memoria da un sito ad un altro dipende da quanto velocemente si modificano le pagine di tale memoria: all'aumentare di D aumenta, quindi, il tempo di migrazione.

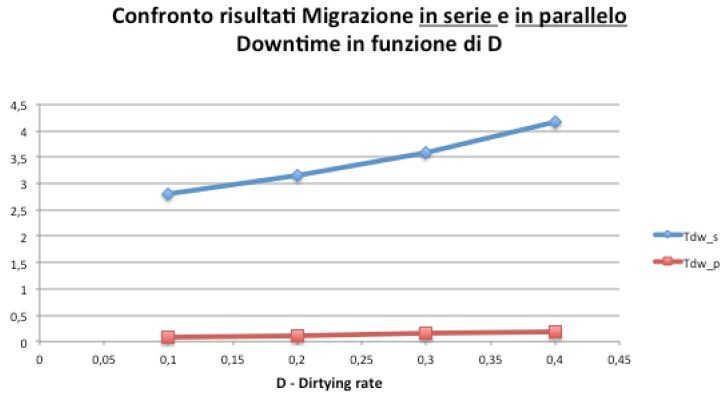


Figura 8.27: Confronto tra il downtime della migrazione in serie e della migrazione in parallelo.

Anche il *downtime* aumenta in funzione di D, ma quello della migrazione *in*

parallelo è significativamente minore rispetto a quello ottenuto dalla migrazione *in serie*, come dimostra Figura 8.27.
Per poter meglio verificare quanto detto sopra, si propone una visione piú accurata dei valori del *downtime* della migrazione in parallelo:

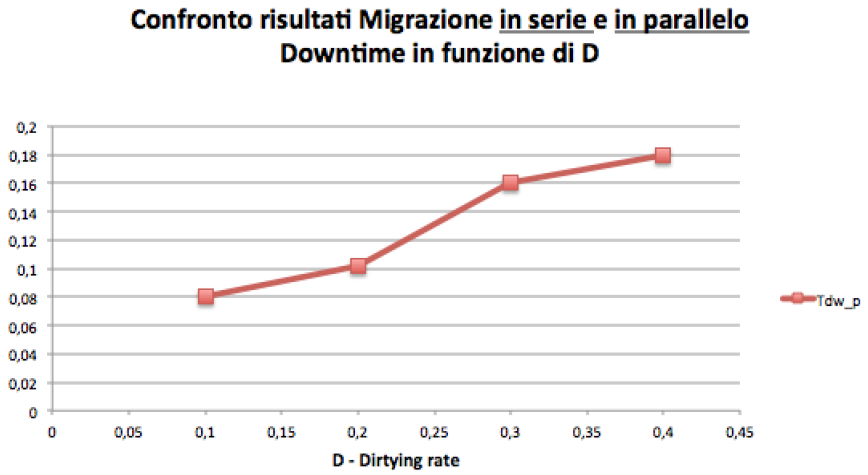


Figura 8.28: Confronto tra il downtime della migrazione in serie e della migrazione in parallelo: particolare della migrazione in parallelo.

Dai risultati di questa comparazione consegue la maggior efficacia della migrazione in parallelo.

8.2.3 Simulazione della migrazione di macchine virtuali aventi diversa dimensione della memoria

Supponiamo ora che le M virtual machine che consideriamo nella migrazione in parallelo possano avere diversa dimensione della memoria. Proprio per questo motivo, durante la migrazione dell'insieme delle M macchine, le VM che hanno dimensione della memoria minore saranno le prime a terminare tale migrazione, e a liberare banda per la trasmissione delle pagine di memoria delle altre macchine.
In Figura 8.29 e Figura 8.30 sono proposti due grafici in qui s'intende rappresentare l'evoluzione temporale del numero di VM ancora coinvolte nella migrazione e del rate di trasmissione.

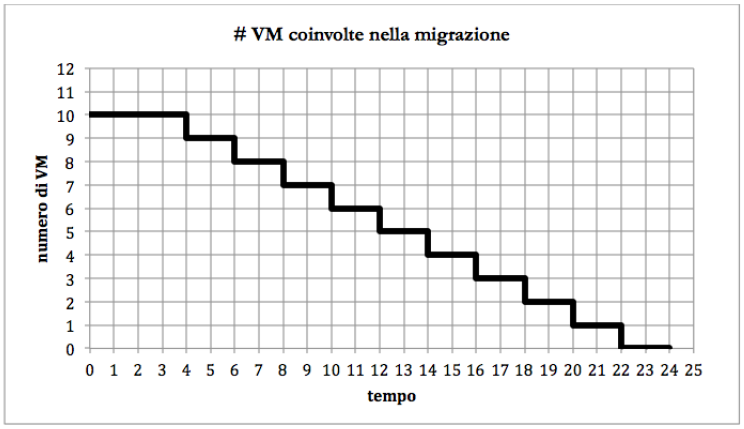


Figura 8.29: Migrazione in parallelo - VM attive.

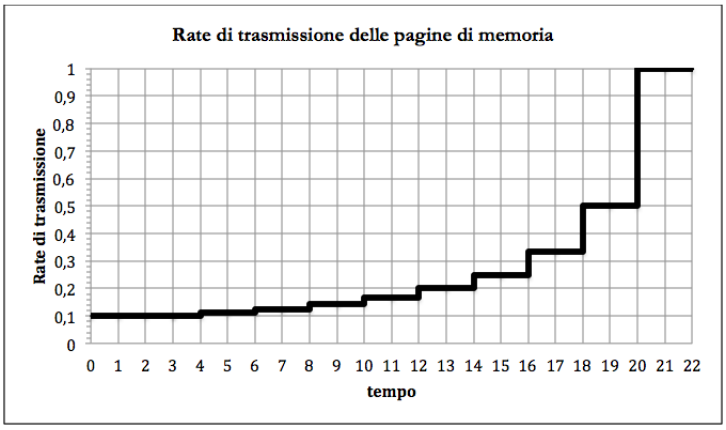


Figura 8.30: Migrazione in parallelo - Rate di trasmissione.

Per calcolare il *downtime* e il *tempo medio di migrazione* è necessario valutare, per ogni iterazione, il *rate di trasmissione*:

$$rt = \frac{R}{nt} \tag{8.7}$$

dove R è il rate di trasmissione e nt è il numero di macchine virtuali ancora coinvolte nella migrazione.
Inoltre, affinché la *dirtying-rate* di una singola macchina virtuale sia sempre la stessa (da noi fissata a 0.5), è importante ricordare di inserire nel simulatore la sua definizione corretta:

$$D = \frac{0.5}{M} \tag{8.8}$$

Per simulare lo scenario proposto occorreranno opportune modifiche al codice (Appendice D). In particolare, in questo ambito, si intenderá il *tempo di attesa medio* come il tempo medio che un set di macchine virtuali deve attendere, partendo dall'istante t=0, prima di poter iniziare la migrazione, e quindi dipenderá dal *tempo medio di migrazione* dei set precedenti come descritto in eq. 8.9:

$$w = T_{mig} * (\frac{N}{M} - 1) \tag{8.9}$$

dove w indica il tempo di attesa medio e T_{mig} il tempo medio di migrazione di un set di M virtual machine. É conseguente a tale definizione che il *tempo di*

attesa medio non dipenda affatto da *rho-class*.
Si intenderanno, inoltre, il *tempo medio di migrazione* e il *downtime medio* come le medie del *tempo di migrazione* e del *downtime* del primo set di macchine.

Per verificare i risultati che si otterranno dal simulatore consideriamo M, che ricordiamo essere il numero di macchine virtuali che possono essere migrate in parallelo, pari a 2, cosí da poter confrontare tali risultati con quelli ricavati con l'approccio analitico e riportati in tabella 8.11.
Unicamente per verifica e confronto s'impone, momentaneamente, $V_0 = 5$ e $V_1 = 12$.

$t = 0$ $V_0 = Res_0 = 5$ $Tres_0 = 10$	 $V_1 = Res_1 = 12$ $Tres_1 = 24$
$t = t + min\{Tres\} = 10$ $Res_0 = 0$ $Vm_0 = 2.5 = Res_0$ $Tres_0 = 5$	 $Res_1 = 7$ $iterazioni = 1$ $Tres_1 = 14$
$t = 15$ $Res_0 = 0$ $Vm_0 = 1.25 = Res_0$ $Tres_0 = 2.5$	 $Res_1 = 4.5$ $iterazioni = 2$ $Tres_1 = 9$
$t = 17.5$ $Res_0 = 0$ $Vm_0 = 0.625 = Res_0$ $Tres_0 = 1.25$	 $Res_1 = 3.25$ $iterazioni = 3$ $Tres_1 = 6.5$
$t = 18.75$ $Res_0 = 0$ $Vm_0 = 0.3125 = Res_0$ $Tres_0 = 0.625$ $t = 19.375$	 $Res_1 = 2.625$ $iterazioni = 4$ $Tres_1 = 5.25$

$Res_0 = 0$ $Vm_0 = 0.15625 = Res_0$ $Tres_0 = 0.3125$	$Res_1 = 2.3125$ $iterazioni = 5$ $Tres_1 = 4.625$
$t = 19.6875$ $Res_0 = 0$ $Vm_0 = 0.078125 = Res_0$ $Tres_0 = xx$	 $Res_1 = 2.15625$ $iterazioni = 6$ $Tres_1 = 2,15625$
$T_{downstart} = 19.6875$ $t = 21.84375$ $Res_0 = xx$ $Vm_1 = 5,4609 = Res_1$ $Tres_0 = xx$	 $Res_1 = 0$ $iterazioni = 7$ $Tres_1 = 5,4609$
$t = 27.03468$ $Res_0 = xx$ $Vm_1 = 1.3652 = Res_1$ $Tres_0 = xx$	 $Res_1 = 0$ $iterazioni = 8$ $Tres_1 = 1.3652$
$t = 28.6699$ $Res_0 = xx$ $Vm_1 = 0.3413 = Res_1$ $Tres_0 = xx$	 $Res_1 = 0$ $iterazioni = 9$ $Tres_1 = 0.3413$
$t = 29,0112$ $Res_0 = xx$ $Vm_1 = 0.0853 = Res_1$ $Tres_0 = xx$ $t = 29.0965$ $iterazioni = 11$	 $Res_1 = 0$ $iterazioni = 10$ $Tres_1 = 0.0853$ $END - MIGRATION$

Tabella 8.11: Indici di prestazione ottenuti con il modello analitico

Quindi, gli indici di prestazione saranno:

- $T_{dw} = T_{mig} - T_{downstart} = 29.0965 - 19.6875 = 9.409$;

- $T_{mig}^{medio} = \frac{T_{mig}}{M} = 14.5482$;

- $numero\ medio\ d'iterazioni = \frac{11}{M} = 5.5$.

Si considereranno i valori:

- $V_{th} = 0.1$,
- $seed = 1$,
- $N - samples = 2$,
- $N - classes = 1$,
- $rho - class = 0.1$,
- $serv - class = 1$.

Si indicherá con nt il numero di macchine virtuali attive in un certo istante: il valore di tale variabile è inizializzato a M, il cui valore é, come precedentemente annunciato, pari a 2.
Con il simulatore si ottengono gli stessi risultati di Tabella 8.11:

```
# Tdownstart 19.687500

 # Tmig 29.096558

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                14.548279               9.409058                5.500000

# x 11.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                14.548279               9.409058                5.500000
```

Figura 8.31: Indici di prestazione - simulazione migrazione di VM con diversa dimensione della memoria.

Si considerano ancora M=2 macchine virtuali in un set, ma questa volta si genera casualmente la dimensione della memoria delle macchine, facendo in

modo che il valore medio di tali memorie rimanga, in media, quello dell'esempio precedente in modo tale da poter paragonare i risultati:

$$Valor\ medio = \frac{5+12}{2} = 8.5, \tag{8.10}$$

A tal proposito, si è corretto il simulatore come segue:

```
for (int i=0; i<M; i++) {
     V[i]=rand()%16 +1; //genero un numero casuale tra 1 e 17
}
```

I risultati piú significativi delle simulazioni sono presentati da Figura 8.32 a Figura 8.38:

```
 # V[j] 3.000000

 # V[j] 14.000000

# Tdownstart 11.625000

 # Tmig 26.390869

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0            0.000000                13.195435               14.765869               5.000000

# x 10.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                13.195435               14.765869               5.000000
```

Figura 8.32: Risultati simulazione - dimensioni della memoria generate casualmente tra 1 e 17 unitá.

```
 # V[j] 13.000000

 # V[j] 15.000000

# Tdownstart 51.796875

 # Tmig 54.521942

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0            0.000000                27.260971               2.725067                7.500000

# x 15.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                27.260971               2.725067                7.500000
```

Figura 8.33: Risultati simulazione - dimensioni della memoria generate casualmente tra 1 e 17 unitá.

```
 # V[j] 2.000000

 # V[j] 3.000000

# Tdownstart 7.750000

 # Tmig 9.154297

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                4.577148                1.404297                5.000000

# x 10.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                4.577148                1.404297                5.000000
```

Figura 8.34: Risultati simulazione - dimensioni della memoria generate casualmente tra 1 e 17 unitá.

```
 # V[j] 7.000000

 # V[j] 14.000000

# Tdownstart 27.781250

 # Tmig 37.178421

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                18.589211               9.397171                6.500000

# x 13.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                18.589211               9.397171                6.500000
```

Figura 8.35: Risultati simulazione - dimensioni della memoria generate casualmente tra 1 e 17 unitá.

```
 # V[j] 15.000000

 # V[j] 15.000000

# Tdownstart 59.765625

 # Tmig 59.824219

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                29.912109               0.058594                8.500000

# x 17.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                29.912109               0.058594                8.500000
```

Figura 8.36: Risultati simulazione - dimensioni della memoria generate casualmente tra 1 e 17 unitá.

```
 # V[j] 2.000000

 # V[j] 16.000000

# Tdownstart 7.750000

 # Tmig 26.474121

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                13.237061               18.724121               5.000000

# x 10.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                13.237061               18.724121               5.000000
```

Figura 8.37: Risultati simulazione - dimensioni della memoria generate casualmente tra 1 e 17 unitá.

```
 # V[j] 2.000000

 # V[j] 8.000000

# Tdownstart 7.750000

 # Tmig 15.817871

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                7.908936                8.067871                5.000000

# x 10.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                7.908936                8.067871                5.000000
```

Figura 8.38: Risultati simulazione - dimensioni della memoria generate casualmente tra 1 e 17 unitá.

Com' è possibile notare osservando i risultati proposti, il *tempo medio di migrazione* e il *downtime* dipendono fortemente dalle dimensioni della memoria della macchine virtuali e, soprattutto, dalla loro differenza.

Si riportano, in Figura 8.39, gli indici di prestazione ottenuti simulando la migrazione di 10000 macchine virtuali, aventi dimensione della memoria generata casualmente tra 1 e 101 (per generalitá), prese 10 alla volta (M=10).

```
# Tdownstart 158.750000

 # Tmig 708.991379

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             7.082824                70.899138               550.241379              1.008000

# x 10080.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                7082.823873             70.899138               550.241379              1.008000
```

Figura 8.39: Indici di prestazione - simulazione migrazione di VM con diversa dimensione della memoria per M=10 e N-samples = 10000.

8.2.4 Confronto tra i processi di migrazione

Con l'intento di determinare il processo che consente la migrazione piú efficace di reti di M macchine virtuali, si studiano gli andamenti del *tempo medio di migrazione* e del *downtime* anche di insiemi di macchine aventi diversa dimensione della memoria in funzione della dimensione delle memorie, della soglia, V_{th}, del rate di trasmissione delle pagine, R, e della *dirtying-rate*, D.

Decidiamo di attribuire alle memorie i valori 5 (unitá) e 6 (unitá): $V_0 = 5$, $V_1 = 6$; e impostiamo nel simulatore (Appendice D) i seguenti valori per i parametri proposti:

- $M = 2$;
- $R = 1$;
- $D = 0.25$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.04$;
- $serv - class = 1$;

Facciamo quindi variare il valore di V_{th} tra 0.1 e 0.9 .
I risultati ottenuti sono riportati in Figura 8.40 e in Figura 8.41.

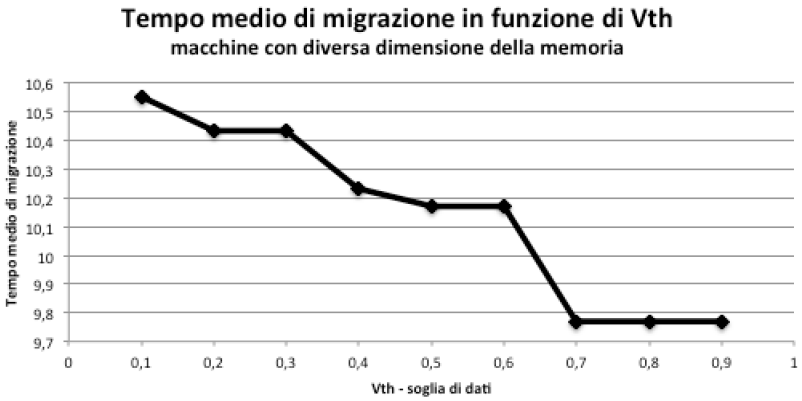


Figura 8.40: Risultati simulazione - VM con diverse dimensioni della memoria - Tempo di migrazione in funzione di V_{th}

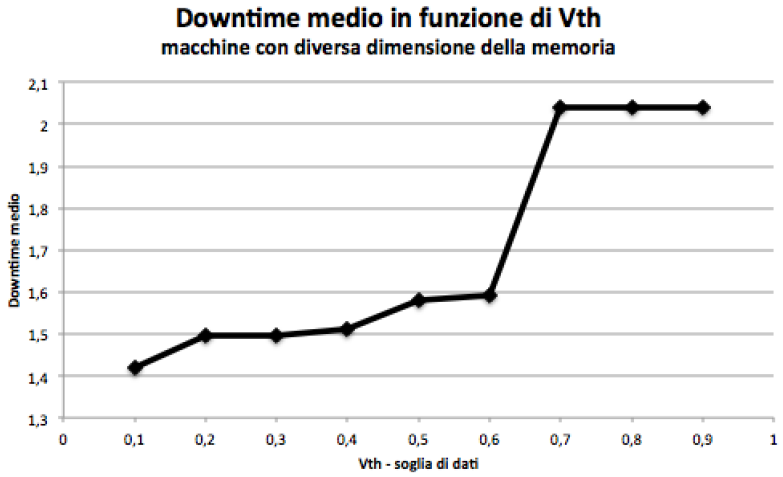


Figura 8.41: Risultati simulazione - VM con diverse dimensioni della memoria - Downtime in funzione di V_{th}

Come atteso, se aumentiamo il valore della soglia di dati che possiamo trasferire durante l'ultima fase, la *stop-and-copy*, diminuisce il tempo medio che s'impiega a trasferire l'intera memoria.

La Figura 8.41 mostra come il *downtime*, invece, aumenti all'aumentare dei dati che si possono trasferire durante la *stop-and-copy*. Tale risultato dipende direttamente dalla definizione di *downtime*.

Ora, impostiamo nel simulatore i valori sotto riportati con l'intento di graficare l'andamento del *tempo medio di migrazione* e del *downtime* in funzione delle dimensioni delle memorie delle macchine coinvolte:

- $M = 2$;
- $V_{th} = 0.1$;
- $R = 1$;
- $D = 0.25$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.04$;
- $serv - class = 1$;

Decidiamo di considerare le seguenti coppie di valori:

- $V_0 = 1$ e $V_1 = 8$;
- $V_0 = 5$ e $V_1 = 12$;
- $V_0 = 6$ e $V_1 = 9$;
- $V_0 = 10$ e $V_1 = 11$.

Ció che si è ottenuto è presentato in Figura 8.42 e Figura 8.43.

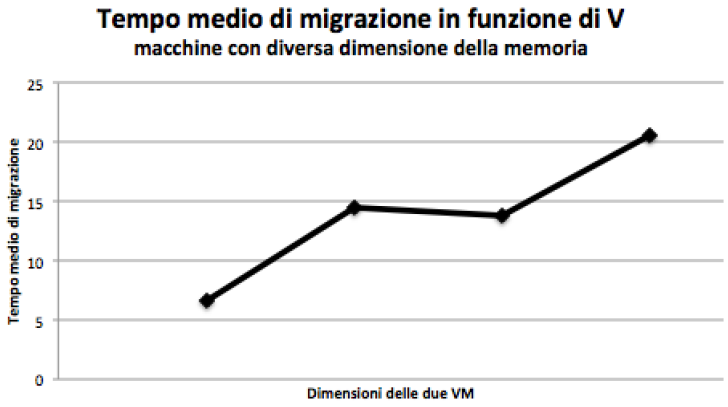


Figura 8.42: Risultati simulazione - VM con diverse dimensioni della memoria - Tempo di migrazione in funzione di V

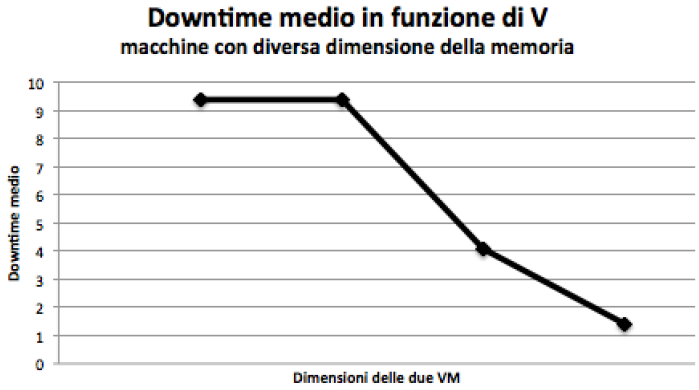


Figura 8.43: Risultati simulazione - VM con diverse dimensioni della memoria - Downtime in funzione di V

In Figura 8.42 si nota come il *tempo di migrazione* dipenda dalle dimensioni delle VM.
In Tabella 8.12 sono riportati i valori esatti ottenuti con il simulatore.

$V_0 = 1$ e $V_1 = 8$	$T_{mig} = 6.57$
$V_0 = 5$ e $V_1 = 12$	$T_{mig} = 14.54$
$V_0 = 6$ e $V_1 = 9$	$T_{mig} = 13.87$
$V_0 = 10$ e $V_1 = 11$	$T_{mig} = 20.55$

Tabella 8.12: Tempo medio di migrazione in funzione di V

Il *tempo medio di migrazione*, dunque, aumenta all'aumentare delle dimensioni delle memorie, ed è tanto maggiore quanto piú è elevata la loro differenza.

Anche per il *downtime* si propone una Tabella in cui sono meglio evidenziati i valori numerici dell'indice di prestazione:

$V_0 = 1$ e $V_1 = 8$	$T_{dw} = 9.4$
$V_0 = 5$ e $V_1 = 12$	$T_{dw} = 9.4$
$V_0 = 6$ e $V_1 = 9$	$T_{mig} = 4.1$
$V_0 = 10$ e $V_1 = 11$	$T_{mig} = 1.42$

Tabella 8.13: Downtime medio in funzione di V

Come è possibile notare in entrambe Figura 8.43 e Tabella 8.13, il *downtime* dipende strettamente dalla differenza tra le dimensioni delle memorie considerate.

A questo punto, fissiamo nel simulatore:

- $M = 2$;
- $V_{th} = 0.1$;
- $V_0 = 5$ e $V_1 = 6$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;

- $rho - class = 0.04$;
- $serv - class = 1$;

e facciamo variare R da 0.5 a 1.
Ricordando che deve sempre valere $M \cdot D < R$, si sceglie $D = 0.2$.
Aggiungiamo, quindi, un ulteriore parametro che rimarrá fisso nel simulatore:

- $D = 0.2$.

Quanto ottenuto dalla simulazione è proposto in Figura 8.44 e Figura 8.45.

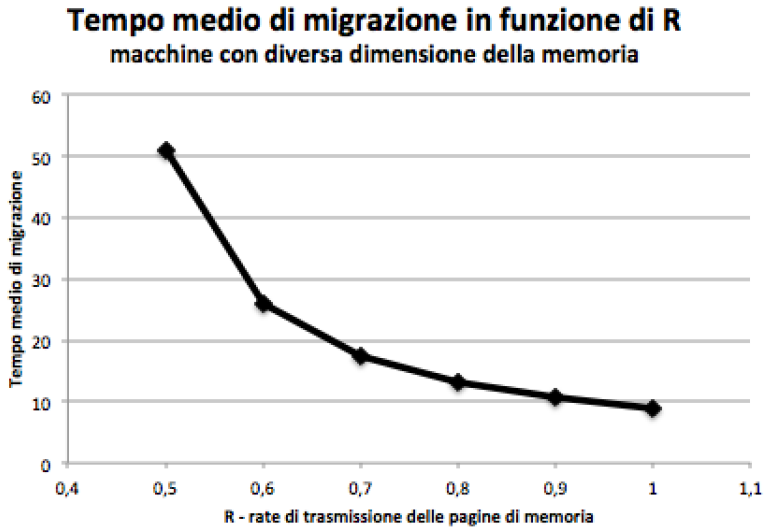


Figura 8.44: Risultati simulazione - VM con diverse dimensioni della memoria - Tempo di migrazione in funzione di R

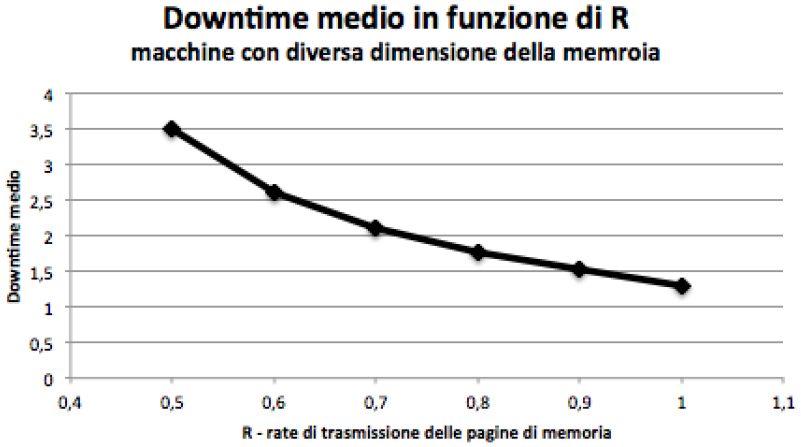


Figura 8.45: Risultati simulazione - VM con diverse dimensioni della memoria - Downtime in funzione di R

In entrambe le figure si osserva come al crescere del rate con cui si trasferiscono le pagine diminuisca sia il *tempo medio di migrazione*, che il *downtime*.

Si presenta, infine, l'andamento del *tempo medio di migrazione* e del *downtime* in funzione della *dirtying-rate*. A tal proposito, si è impostato nel simulatore:

- $M = 2$;
- $V_0 = 5$ e $V_1 = 6$;
- $R = 1$;
- $V_{th} = 0.1$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.04$;
- $serv - class = 1$;

e si è fatto variare D tra 0.1 e 0.4 .
Riportiamo in Figura 8.46 e in Figura 8.47 quando ottenuto dal simulatore.

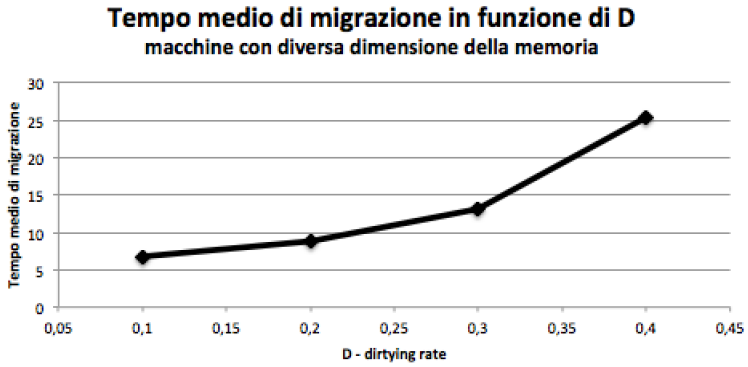


Figura 8.46: Risultati simulazione - VM con diverse dimensioni della memoria - Tempo di migrazione in funzione di D

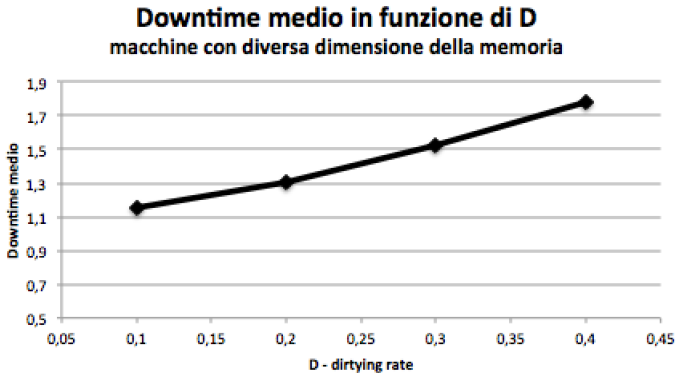


Figura 8.47: Risultati simulazione - VM con diverse dimensioni della memoria - Downtime in funzione di D

Ancora una volta, i risultati rispecchiano l'attesa. Infatti, maggiore è il tasso con cui si modificano le pagine delle memorie, maggiore è il tempo impiegato per avere una copia coerente delle memorie sul sito destinazione della migrazione.

Cerchiamo, ora, di capire quanto incide sulla migrazione *in parallelo* avere a che fare con macchine che hanno diversa dimensione della memoria.
Ricordiamo, innanzitutto, che il tempo medio di migrazione nel caso in cui le macchine hanno la medesima dimensione è definito in maniera diversa dal caso in cui le macchine presentano dimensioni differenti. In particolare, la seconda situazione prevede che le M macchine di un insieme presentino la *richiesta di migrazione* contemporaneamente; nella prima, invece, ció non è affatto vero: ogni macchina fa la richiesta dopo un certo *expon(lambda)* dalla richiesta della macchina precedente, con *expon(lambda)* che presenta distribuzione esponenziale.

```
double expon(double param) {
// Declare some useful variables
int rnd_num;
double unif,val;

// Generate a uniform random number between 0 and 1
rnd_num = rand();
if (rnd_num == RAND_MAX) rnd_num--;
unif = (double)rnd_num/RAND_MAX;

// Transform the uniform random variable into
an exponential one using the inverse function rule
val = -log(1.0-unif)/param;

return val;

}
...
if (tot_arrivals < N) {
insert_new_event(now + expon(lambda[e->pclass]),
MIGRATION_REQ,e->pclass,Vmem, 0, 0, 0, 0);
}
```

Impostiamo, in entrambi i simulatori, i seguenti valori:

- $M = 2$;
- $V_0 = 5$ e $V_1 = 5$;

- $V_{th} = 0.1$;
- $R = 1$;
- $D = 0.25$;
- $seed = 1$;
- $N - samples = 2$;
- $N - classes = 1$;
- $rho - class = 0.04$;
- $serv - class = 1$;

I risultati sono proposti in Figura 8.48 e Figura 8.49.

```
# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                9.921875                0.078125                6.500000

# Carico effettivo 0.396875

# x 13.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                9.921875                0.078125                6.500000
```

Figura 8.48: Risultati simulazione - VM con identiche dimensioni della memoria - Simulatore Appendice C

```
# Tdownstart 19.687500

 # Tmig 19.765625

# Class         Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
  0             0.000000                9.882812                0.078125                6.500000

# x 13.000000

# Overall:      Mean waiting time       Mean migration time     Mean downtime   Mean number of iterations
                0.000000                9.882812                0.078125                6.500000
```

Figura 8.49: Risultati simulazione - VM con uguali dimensioni della memoria - Simulatore Appendice D

In effetti, ció per cui differiscono i due simulatori, una volta forzate $V_0 = 5$ e $V_1 = 5$, è solamente il *tempo medio di migrazione*.

Verifichiamo, a questo punto, quanto influenza il *tempo medio di migrazione* e il *downtime* il fatto di aver memorie di dimensione diversa all'interno di uno stesso insieme.
A tale scopo, generiamo memorie con dimensioni comprese tra 1 e 10, in modo da ottenere lo stesso valor medio del caso di VM identiche:

$$Valor - medio = \frac{5+5}{2} = 5, \tag{8.11}$$

Si corregge il simulatore come segue:

```
for (int i=0; i<M; i++) {
     V[i]=rand()%9 +1; //genero un numero casuale tra 1 e 10
}
```

In Tabella 8.14 e Tabella 8.15 sono presentati i risultati della valutazione proposta:

$V_0 = 5$ e $V_1 = 5$	$T_{mig} = 9.92$	$T_{dw} = 0.078$

Tabella 8.14: Tempo di migrazione e downtime - Simulatore Appendice C

$V_0 = 5$ e $V_1 = 8$	$T_{mig} = 11.88$	$T_{dw} = 4.07$
$V_0 = 2$ e $V_1 = 3$	$T_{mig} = 4.57$	$T_{dw} = 1.40$
$V_0 = 4$ e $V_1 = 5$	$T_{mig} = 8.57$	$T_{dw} = 1.41$
$V_0 = 2$ e $V_1 = 2$	$T_{mig} = 3.90$	$T_{dw} = 0.062$
$V_0 = 1$ e $V_1 = 8$	$T_{mig} = 6.57$	$T_{dw} = 9.40$

Tabella 8.15: Tempo di migrazione e downtime - Simulatore Appendice D - Diverse memorie

Come è evidente dal confronto delle tabelle, gli indici di prestazione della migrazione sono dipendenti dalle dimensioni delle memorie e, soprattutto, dalla loro differenza.
Per questo motivo conviene, se possibile, raggruppare in un *set* macchine aventi memorie uguali, o molto simili, cosí da ottenere *downtime* e *tempo medio di*

migrazione poco elevati.

Dalle analisi e dai confronti operati per i processi di migrazione *in serie*, *in parallelo*, e *in parallelo con dimensione delle memorie diverse*, risulta piú vantaggiosa ed efficiente la scelta di trasferire *in parallelo* le virtual machine, avendo cura, dove possibile, di raggruppare in un *insieme* quelle macchine che hanno la stessa dimensione della memoria, o dimensioni simili.

Capitolo 9

Impatto dei parametri di sistema sugli indici di prestazione

Dai risultati ottenuti dalle comparazioni di paragrafo 8.2.4 abbiamo osservato che il *tempo di migrazione* e il *downtime* dipendono dalla differenza delle dimensioni delle memorie.
Analizziamo, allora, la *varianza* e grafichiamo gli andamenti di T_{mig} e T_{dw} in funzione di questo parametro.

9.1 Valor medio e varianza di una variabile aleatoria

Sia V_i la dimensione della memoria dell'i-esima macchina virtuale, $\forall i = 1, ..., M$.
Si definisce *valor medio* della dimensione della memoria, e si indica con $\mathbb{E}[V]$, la grandezza:

$$\mathbb{E}[V] = \frac{\sum_{i=1}^{M} V_i}{M} \tag{9.1}$$

Sia $var(V)$ la *varianza* della dimensione della memoria, definita come:

$$var(V) = \mathbb{E}[(V - \mathbb{E}[V])^2] = \frac{\sum_{i=1}^{M} (V_i - \mathbb{E}[V])^2}{M} \tag{9.2}$$

9.2 Tempo di migrazione e downtime in funzione della varianza

Scegliamo, per semplicitá, $M = 2$ e andiamo a calcolare la *varianza* delle seguenti coppie di dimensioni delle memorie.

1. $V_0 = 45$ *e* $V_1 = 55$
2. $V_0 = 40$ *e* $V_1 = 60$
3. $V_0 = 35$ *e* $V_1 = 65$
4. $V_0 = 30$ *e* $V_1 = 70$
5. $V_0 = 25$ *e* $V_1 = 75$
6. $V_0 = 20$ *e* $V_1 = 80$
7. $V_0 = 15$ *e* $V_1 = 85$
8. $V_0 = 10$ *e* $V_1 = 90$
9. $V_0 = 5$ *e* $V_1 = 95$
10. $V_0 = 1$ *e* $V_1 = 99$

Si ottiene:

Inserendo nel simulatore presentato in Appendice D le coppie di valori di memorie sopra elencate, e i valori:

- $M = 2$
- $D = 0.25$
- $R = 1$
- $seed = 1$
- $N - samples = 2$
- $N - classes = 1$

$\mathbb{E}[V] = \frac{55+45}{2} = 50$	$var(V) = \frac{(45-50)^2+(55-50)^2}{2} = 25$
$\mathbb{E}[V] = \frac{60+40}{2} = 50$	$var(V) = \frac{(40-50)^2+(60-50)^2}{2} = 100$
$\mathbb{E}[V] = \frac{65+35}{2} = 50$	$var(V) = \frac{(35-50)^2+(65-50)^2}{2} = 225$
$\mathbb{E}[V] = \frac{70+30}{2} = 50$	$var(V) = \frac{(30-50)^2+(70-50)^2}{2} = 400$
$\mathbb{E}[V] = \frac{75+25}{2} = 50$	$var(V) = \frac{(25-50)^2+(75-50)^2}{2} = 625$
$\mathbb{E}[V] = \frac{80+20}{2} = 50$	$var(V) = \frac{(20-50)^2+(80-50)^2}{2} = 900$
$\mathbb{E}[V] = \frac{85+15}{2} = 50$	$var(V) = \frac{(15-50)^2+(85-50)^2}{2} = 1225$
$\mathbb{E}[V] = \frac{90+10}{2} = 50$	$var(V) = \frac{(10-50)^2+(90-50)^2}{2} = 1600$
$\mathbb{E}[V] = \frac{95+5}{2} = 50$	$var(V) = \frac{(5-50)^2+(95-50)^2}{2} = 2025$
$\mathbb{E}[V] = \frac{99+1}{2} = 50$	$var(V) = \frac{(1-50)^2+(99-50)^2}{2} = 2401$

Tabella 9.1: Valor medio e varianza di V

- $rho - class = 0.04$
- $serv - class = 1$

si ottengono i valori del *tempo medio di migrazione* e del *downtime* proposti in Tabella 9.2.

$V_0 = 45$ *e* $V_1 = 55$	$T_{mig} = 96.53$	$T_{dw} = 13.42$	numero d'iterazioni = 16
$V_0 = 40$ *e* $V_1 = 60$	$T_{mig} = 93.22$	$T_{dw} = 26.75$	numero d'iterazioni = 15
$V_0 = 35$ *e* $V_1 = 65$	$T_{mig} = 89.90$	$T_{dw} = 40.07$	numero d'iterazioni = 16
$V_0 = 30$ *e* $V_1 = 70$	$T_{mig} = 86.58$	$T_{dw} = 53.40$	numero d'iterazioni = 16
$V_0 = 25$ *e* $V_1 = 75$	$T_{mig} = 83.19$	$T_{dw} = 66.78$	numero d'iterazioni = 15
$V_0 = 20$ *e* $V_1 = 80$	$T_{mig} = 79.89$	$T_{dw} = 80.09$	numero d'iterazioni = 15
$V_0 = 15$ *e* $V_1 = 85$	$T_{mig} = 76.58$	$T_{dw} = 93.40$	numero d'iterazioni = 15
$V_0 = 10$ *e* $V_1 = 90$	$T_{mig} = 73.22$	$T_{dw} = 106.76$	numero d'iterazioni = 14
$V_0 = 5$ *e* $V_1 = 95$	$T_{mig} = 69.89$	$T_{dw} = 120.09$	numero d'iterazioni = 13
$V_0 = 1$ *e* $V_1 = 99$	$T_{mig} = 67.23$	$T_{dw} = 130.71$	numero d'iterazioni = 10

Tabella 9.2: Tempo di migrazione e downtime in funzione della varianza

Come si può osservare dalla tabella, sia il *tempo medio di migrazione* che il *downtime* dipendono dalla varianza: all'aumentare della differenza tra le

dimensioni delle memorie diminuisce T_{mig}, viceversa, cresce T_{dw}.

Di seguito si propongono due grafici: nel primo è mostrato l'andamento del *tempo medio di migrazione* in funzione della *varianza*:

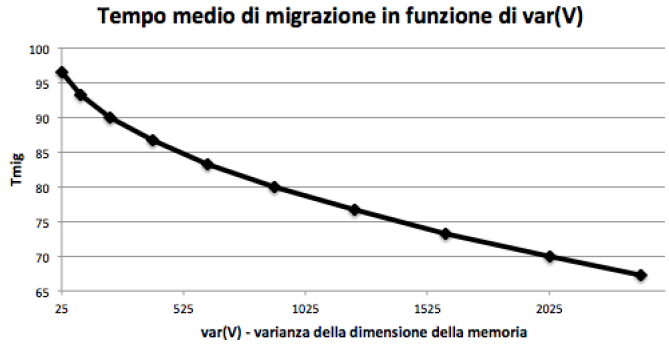


Figura 9.1: Tempo medio di migrazione in funzione della varianza.

nel secondo l'andamento del *downtime*, sempre in funzione della *varianza*:

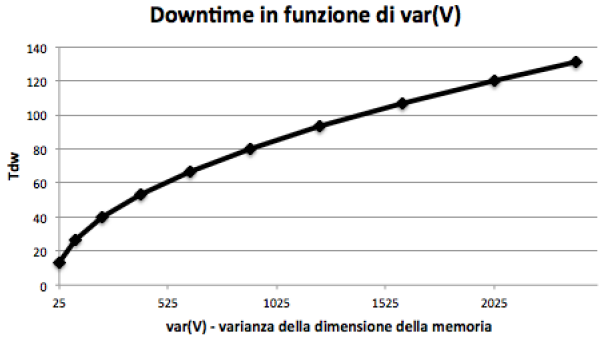


Figura 9.2: Downtime in funzione della varianza.

Cerchiamo, a questo punto, di capire perché il *tempo medio di migrazione* diminuisce, mentre è il contrario per il *downtime*.

Si pensi che più due memorie (qui si considera il caso $M = 2$, ma può essere generalizzato) sono simili in dimensioni, più saranno simili i tempi impiegati nella loro migrazione. Perció, la banda sará suddivisa tra tutti gli utenti per quasi tutto il tempo. Da qui derivano i *tempi medi di migrazione* elevati delle

prime coppie di memorie.
Invece, più le dimensioni delle memorie sono diverse (una molto bassa e una molto elevata), prima si libera la banda da dedicare completamente ad un solo utente, e quindi i *tempi medi di migrazione* saranno più contenuti.

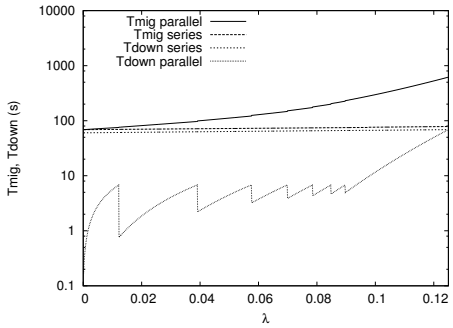
Il *tempo di downtime*, invece, è definito come la differenza tra l'istante in cui termina la migrazione e l'istante in cui la prima macchina entra nella fase di *stop-and-copy*: maggiore è la differenza tra le dimensioni delle memorie, maggiore è questa differenza, quindi, il *tempo di downtime*.

9.3 Valutazioni degli indici di prestazione nel caso di parametri realistici

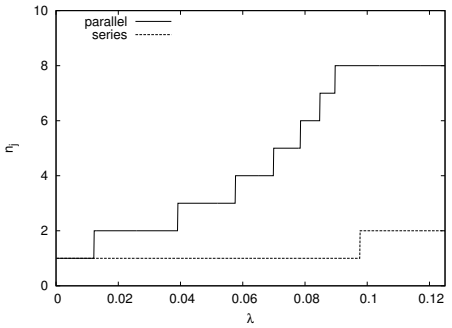
In precedenza si sono considerati valori per la dimensione della memoria, per il *rate di trasmissione*, ecc., per nulla verosimili in quanto l'unico scopo era la creazione e la validazione di un modello.
In questo capitolo si vogliono associare valori più realistici ai parametri in gioco nella migrazione.
Con riferimento alle *edge network* si considera [31]:

- $M = 8$;
- $R = 1Gbps$, bit rate ragionevole per le comunicazioni intra-Data Center;
- $V_i = 1GB$, valore tipico per le attuali tecnologie di virtualizzazione;
- $D = 80Mbps$, corrispondente a $\lambda = 0.08$, è una ragionevole stima della *dirtying-rate* sulla base di misure effettuate su vari benchmark [13];
- $V_{th} = 800Mb$, giusto compromesso considerati gli altri parametri;
- $n_{max} = 8$, numero massimo di iterazioni - è possibile utilizzare questo valore come soglia per limitare il tempo di trasferimento di una VM;

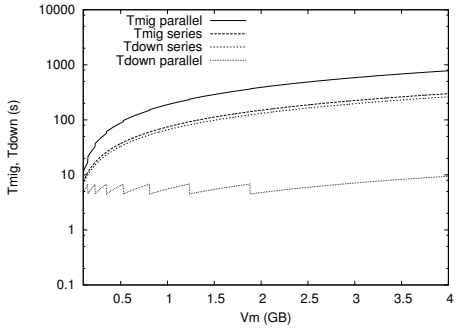
In [31] sono proposti grafici in cui si confrontano gli indici di prestazione della migrazione *in serie* e *in parallelo*, ottenuti impiegando le definizioni presentate in eq. (8.1), (8.2), (8.3), (8.4), (8.5), (8.6), modificando uno alla volta i parametri sopra elencati mantenendo gli altri invariati:



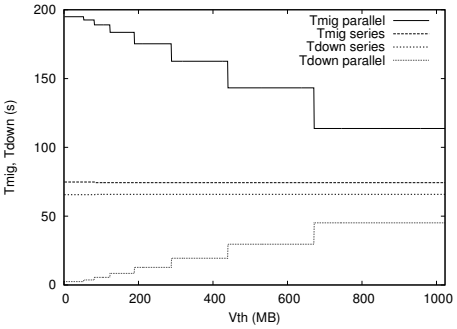
(a) *Indici di prestazione in funzione di* λ.



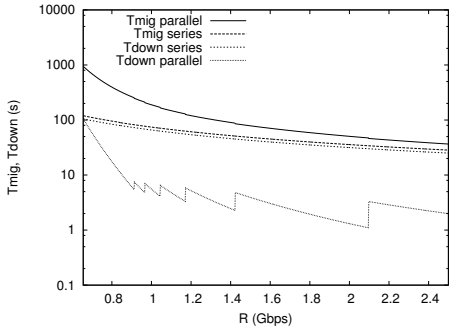
(b) *Numero d'iterazioni in funzione di* λ.



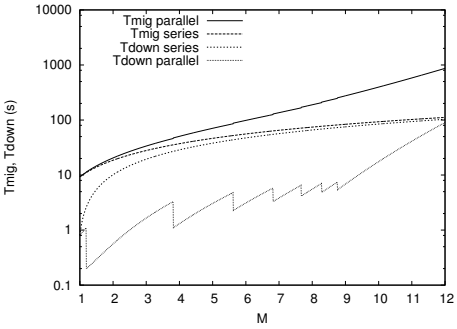
(c) *Indici di prestazione in funzione della dimensione della memoria.*



(d) *Indici di prestazione in funzione della soglia.*



(e) *Indici di prestazione in funzione del rate di trasmissione.*



(f) *Indici di prestazione in funzione del numero di macchine in un set.*

Figura 9.3: Indici di prestazione - valori realistici.

In Figura 9.3(a) compaiono il *tempo medio di migrazione* e il *downtime* in funzione di λ, che ricordiamo essere il rapporto tra la *dirtying-rate* e il *rate di trasmissione*: $\lambda = \frac{D}{R}$. Si nota che per λ maggiore di 0.04 il *tempo medio di migrazione in parallelo* cresce piuttosto velocemente, mentre quello della *serie* non mostra la stessa sensibilità. Ciò è principalmente dovuto al differente numero di iterazioni impiegate dai due processi: in Figura 9.3(b) è proposto l'andamento di n_j in funzione dello stesso λ in cui è evidente come, a causa del ridotto *rate* disponibile alla singola macchina, la migrazione *in parallelo* si concluda con più iterazioni.
Figura 9.3(c) paragona i processi *serie* e *parallelo* in funzione della quantità di memoria allocata ad ogni VM: il *tempo di migrazione* cresce linearmente in entrambi i casi, così come il *downtime* del processo in serie. Il *downtime* della migrazione in parallelo, invece, fluttua attorno ai 5s, e inizia a crescere solo quando si raggiunge n_{max} perché, arrivati a tale soglia, tutti i MB di dati devono essere trasmessi, e questo causa l'aumento del *downtime*.
Il ruolo dell'altra soglia, V_{th}, è illustrato in Figura 9.3(d). Ad un valore basso di V_{th} corrisponde un numero elevato di iterazioni prima della fase di *stop-and-copy* e, quindi, un *tempo medio di migrazione* elevato, mentre un *downtime* piuttosto contenuto. Con i valori di riferimento scelti in [31], si osserva che la migrazione in parallelo è piú sensibile al valore di V_{th} rispetto a quella in serie a causa dell'elevato valore di λ.
In Figura 9.3(e) si nota come gli indici di prestazione decrescano all'aumentare del *rate di trasmissione*. Questa figura puó essere utile per dimensionare la banda necessaria per raggiungere determinati *tempi di migrazione* e *downtime* all'interno di un Data Center, nota la *dirtying-rate*.
Figura 9.3(f) mostra l'impatto che ha la dimensione dell'insieme delle VM sugli indici d'interesse: all'aumentare di M aumentano entrambi T_{mig} e T_{dw} (T_{down} in figura): come si puó osservare, il processo *in parallelo* risente maggiormente della variazione del numero di macchine in un *set* rispetto al processo di migrazione *in serie*.

È comunque bene sottolineare come i *tempi di migrazione* siano, generalmente, più elevati nella migrazione *in parallelo*. Al contrario, il *downtime* risulta minore rispetto alla *serie*.

Capitolo 10

Caratterizzazione della *dirtying-rate*

Finora si è considerata la *dirtying-rate* fissa e costante nonostante questa, in realtá, dipenda fortemente dalle applicazioni in esecuzione sulla VM.

In [13] si sfrutta la capacitá di Xen [32] di tracciare gli accessi alla memoria per misurare tale *dirtying-rate*: si utilizza la modalitá *shadow paging* per ottenere le statistiche sulle pagine che si modificano durante l'esecuzione di differenti workload sulla VM.

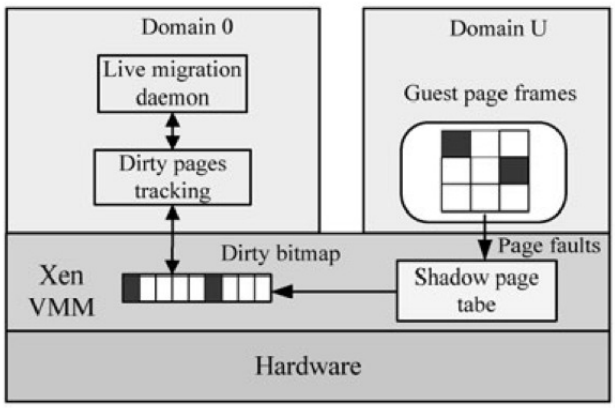


Figura 10.1: Tracciamento delle *dirty-pages* di Xen.

Lo *shadow paging* permette di tenere memoria delle pagine che si modificano durante un determinato intervallo temporale: in [13] si è scelto un intervallo

di 60s, e si sono misurate le *dirtying-rate* per workload come TCP-C, Linux in idle, Dbench, Linpack e SPECWeb2005.
Ogni 0.5s all'interno della finestra dei 60s hanno guardato la bitmap, e poi l'hanno "resetta" ogni volta.

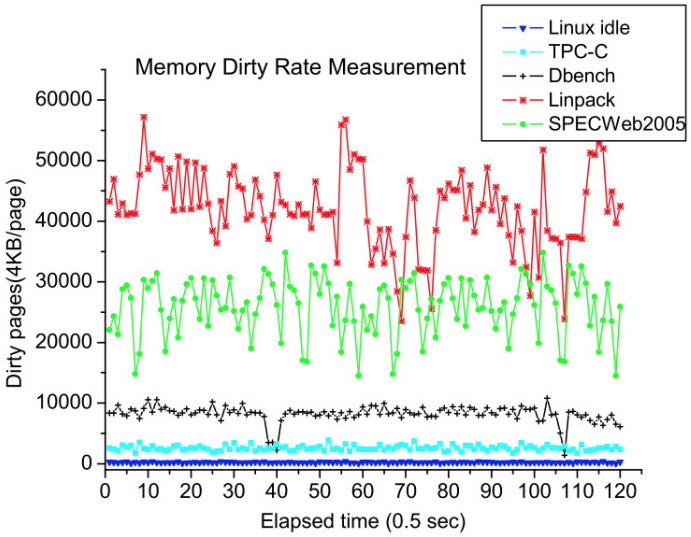


Figura 10.2: Tasso di pagine modificate ogni 0.5s per diversi workload in esecuzione su una VM con 1GB di RAM.

Figura 10.2 riporta il numero di pagine modificate generate da ogni workload in esecuzione sulla VM. Dalle curve proposte si osserva che la *dirtying-rate* varia significativamente tra i workload, ma si nota anche che per alcuni di questi, come Linux in idle, TCP-C e Dbench, tale parametro rimane pressoché costante nel tempo, il che porta a considerare l'assunzione fatta per la costruzione del modello (*dirtying-rate* costante) non del tutto inappropriata.

C'è anche un altro modo che gli autori di [13] usano per misurare la *dirtying-rate*. Propongono di eseguire l'applicazione d'interesse sulla VM e di osservare la bitmap ogni 50ms all'interno della finestra di 60s, ma questa volta senza mai "pulire" la bitmap. Ció permette di misurare il *Writable Working Set* in un intervallo di 60 secondi.

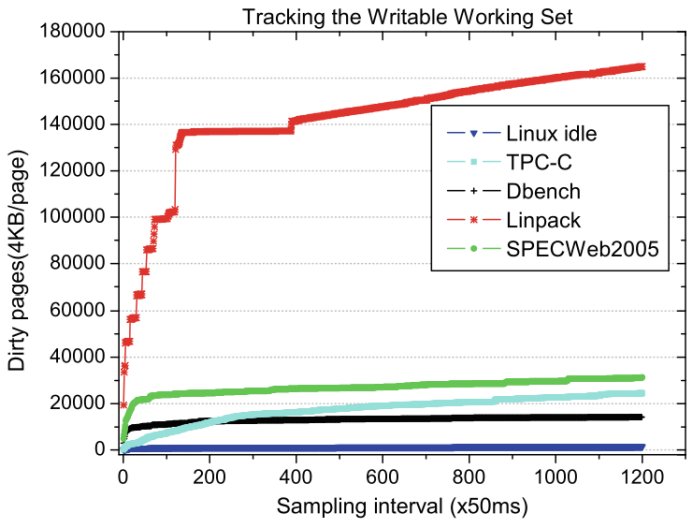


Figura 10.3: WWS dei workload considerati.

In Figura 10.3 è rappresentato il numero di pagine modificate da ogni workload. Si puó notare che la maggior parte di questi esibisce un'elevata *dirtying-rate* all'inizio delle misurazioni (la *dirtying-rate* è la pendenza della retta che congiunge due punti di ogni curva), dopodiché, con il trascorre del tempo d'osservazione, questa diminuisce in tutte le applicazioni. Il motivo è che tutti i workload hanno un insieme di pagine *'hot'* che tendono a modificarsi spesso, ma il loro aggiornamento non viene registrato nel WWS in quanto la bitmap non viene resettata ad ogni osservazione.

10.1 Modello stocastico della gestione della memoria

In [33] è proposta un'analisi matematica per determinare la probabilitá di scrittura dell'i-esima pagina di memoria, che indicheremo con m_i.

Chiamiamo $N_i(t)$ il numero causale di eventi di riscrittura dell'i-esima pagina al tempo t, $\forall i = 1, ..., L$, dove L è il numero totale di pagine di memoria di una VM.
Si assume $N_i(t) = 0$ al tempo $t = 0$, e che la sequenza di ri-scritture nel processo

$N_i(t)$ sia un *Processo puntuale di Poisson* [34]: ció comporta che gli eventi di ri-scrittura della pagina m_i siano indipendenti l'uno dall'altro, che avvengano ad una *dirtying-rate* media fissa, e che l'evento di due ri-scritture contemporanee sia impossibile, con $N_i(t)$ statisticamente indipendente da $N_j(t)$, per $i \neq j$.
Sia D_i la *dirtying-rate* media della pagina m_i e sia $p_i(n)$ la probabilitá che la pagina m_i venga modificata almeno una volta durante l'intervallo di durata $\frac{n}{\delta}$, dove n è il numero medio di pagine trasmesse e δ è il throughput (pagine trasmesse nell'intervallo di tempo), ottenuta come:

$$p_i(n) = 1 - e^{-\frac{D_i \cdot n}{\delta}} \tag{10.1}$$

È da notare che se $D_i \gg \frac{\delta}{n}$, allora $p_i \approx 1$, il che significa che quasi sicuramente la pagina m_i si modificherá durante il trasferimento delle n pagine: questo è il caso in cui il trasferimento della pagina in questione è uno spreco di tempo, banda, e potenza.
D'altro canto, quando $D_i \ll \frac{\delta}{n}$, allora $p_i \approx 0$ e la pagina m_i non sará praticamente mai riscritta in $\frac{n}{\delta}$ secondi.
Alla fine della $k-esima$ iterazione di *pre-copy* la probabilitá $p_i^{(k)}$ che la pagina m_i sia modificata almeno una volta è descritta in eq. 10.2

$$p_i^{(k)} = \begin{cases} p_i(L_k - \phi_k^{-1}(i) + 1), & k = S_i \\ p_i(L_k - \phi_{S_i}^{-1}(i) + 1 + \sum_{l=S_i+1}^{k} M_l), & k > S_i \end{cases} \tag{10.2}$$

dove $S_i \in \{0, 1, ..., k\}$ è l'indice dell'ultima iterazione in cui la pagina m_i è stata ritrasmessa, $\phi_k : \{1, 2, ..., L_k\} \rightarrow \{\phi_k^{(1)}, \phi_k^{(2)}, ..., \phi_k^{(L_k)}\}$ è la funzione di "ordinamento per il trasferimento delle pagine" alla k-esima iterazione, e L_k sono le pagine che vengono trasferite durante quell'iterazione.

Per calcolare il *downtime* e il *tempo medio di migrazione*, comunque, anche Darsena et al. assumono $D_i = D$, ovvero ipotizzano che tutte le L pagine della memoria abbiano la stessa *dirtying-rate*, $\forall i = 1, ..., L$.
I risultati, proposti in Figura 10.4, li hanno ottenuti considerando:

- $R = 1Gbps$
- $L \in \{10, 50, 100\}$
- $V_{th} = \frac{L}{10}$, è la soglia di pagine al di sotto della quale incomincia la *stop-and-copy*

- $K_{max} = 5$, è il numero massimo di iterazioni oltre il quale incomincia la *stop-and-copy*
- $\delta = 28710pps$

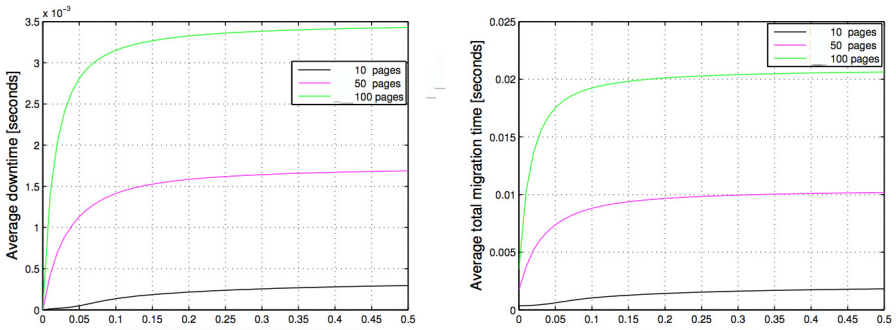


Figura 10.4: Downtime e Tempo di migrazione in funzione di $\frac{D}{\delta}$.

10.2 Propagazione efficiente della memoria

Nei capitoli precedenti si è dimostrato come la *dirtying-rate* influisca sul *tempo di migrazione* e sul *downtime*, che sono i parametri chiave della migrazione *live*.
In [18] gli autori hanno proposto un'approccio, chiamato Microwiper, che permette di calcolare la *dirtying-rate*, e sfrutta questa conoscenza per ridurre i valori degli indici di prestazione fino al 50%.
Microwiper è composto da due strategie: la prima, "ordinamento delle pagine", consiste nel raggruppare le pagine della memoria in *strisce* di lunghezza fissata, quindi nell'ordinarle per dirtying-rate crescente; la seconda,"trasferimento limitato", prevede la non trasmissione della *striscia* con dirtying-rate maggiore durante i turni di *pre-copy*, posticipandone il trasferimento durante la *stop-and-copy*.
Per predirre la *dirtying-rate* con cui si modificano le pagine, Du et al. si basano sul *"principio di localitá temporanea"* che sfrutta la conoscenza di tale *rate* all'inizio dei *rounds* di *pre-copy*.

10.2.1 Ordinamento delle pagine

Il tasso con cui si modifica una pagina di memoria di una VM è diverso per ogni pagina. Quelle che l'hanno elevato hanno la possibilitá di essere modificate nuovamente molto alta, e trasmetterle vorrebbe dire modificarle e trasmetterle ancora. Al contrario, le pagine che hanno dirtying-rate bassa sono buone candidate per il trasferimento.
In [18] si decide, allora, di ordinare le pagine per *rate* crescente e, successivamente, di trasferirle seguendo il nuovo ordine.

Siccome gli autori sottolineano la difficoltá di misurare con esattezza la dirtying-rate in tempo reale, che invece è la base per poter applicare Microwiper, propongono di raggruppare le pagine in *strisce* prima di ordinarle e trasferirle. Il beneficio sta nell'avere un tasso di scrittura piú stabile. Per calcolarlo, Microwiper conta il numero di pagine modificate per ogni *striscia* ad ogni iterazione, divide il numero per la durata di quell'iterazione, e ottiene cosí la *dirtying-rate* di una *striscia*.

10.2.2 Trasferimento limitato

Ricordando che il *Writable Working Set* è l'insieme delle pagine che si modificano piú spesso, è fondamentale identificarlo e trasmetterlo durante la fase finale (*stop-and-copy*).
La strategia del "trasferimento limitato" si basa proprio sul posticipare la trasmissione della *striscia* con la piú alta *dirtying-rate* durante la *stop-and-copy*.

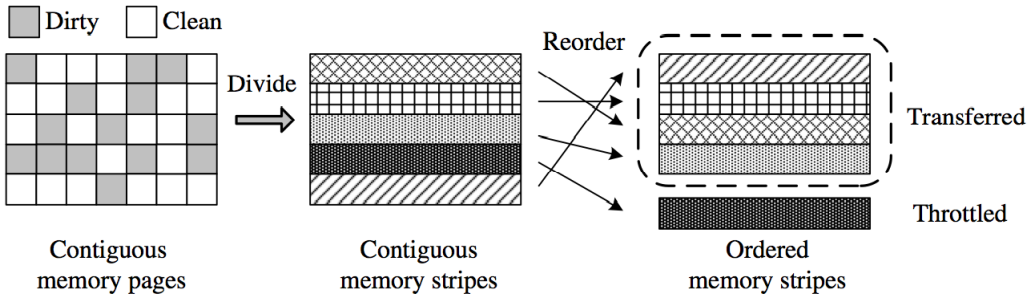


Figura 10.5: Strategie di Microwiper.

Du et al. considerano due workload per verificare l'efficacia dell'approccio proposto: il primo è *kernel compiling*, che rappresenta i programmi i cui accessi alla memoria sono relativamente fissi e costanti; il secondo è *bzip2 decompressor* (bunzip2), che rappresenta quei programmi in cui la scrittura della memoria è molto frequente.
In Figura 10.6 è mostrata un'istantanea della memoria della VM per ogni programma (i segni + indicano una *dirty page*).

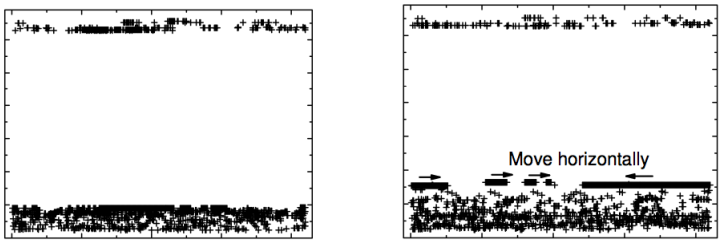


Figura 10.6: Istantanea della bitmap della memoria di una VM.

In Figura 10.6(a) si puó osservare che il *kernel compiling* presenta pochi accessi alla memoria: le pagine che si modificano sono, infatti, sempre le stesse (buon principio di localitá temporanea). Al contrario, bunzip2 mostra molte "tracce" degli accessi alla memoria: possono apparire ovunque insiemi di *dirty pages*, come staticamente raffigurato in Figura 10.6(b).

Sono utilizzate tre metriche per comparare Microwiper e XLM (Xen Live Migration) in [18]: il *downtime*, il *tempo di migrazione*, e il numero extra di pagine trasferite (pagine trasferite sottratte alle pagine totali della memoria). Per determinare i valori di tali parametri gli autori eseguono 15 migrazioni e calcolano la media aritmetica per ogni metrica. Le misurazioni sono effettuate su tre VM aventi dimensioni della memoria pari a 256MB, 512MB e 1024MB.

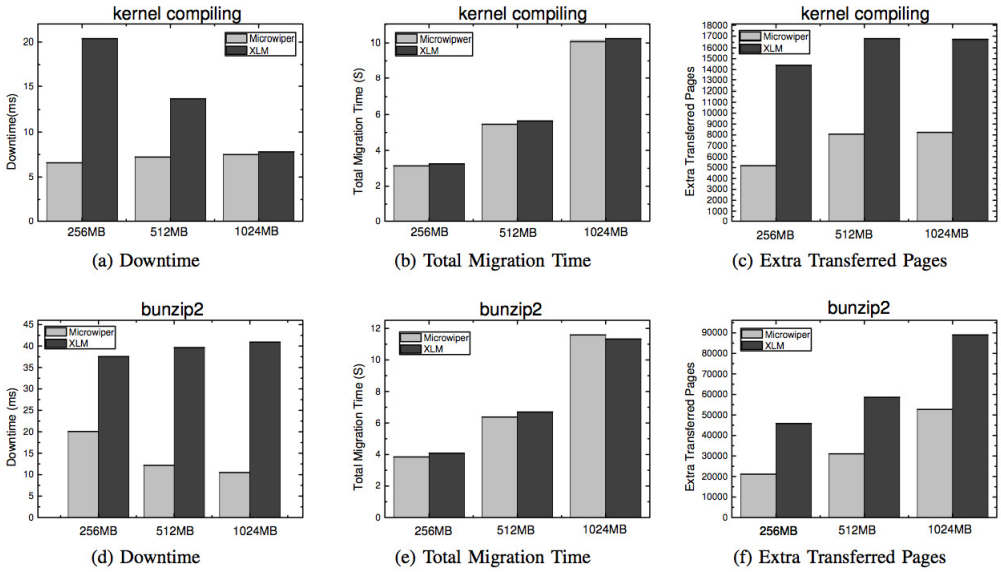


Figura 10.7: Confronto tra Microwiper e XLM.

Come si nota da Figura 10.7, per entrambi i workload, XLM prevede piú iterazioni e propaga piú *dirty pages* per ogni iterazione, il che giustifica i tempi di migrazione e downtime piú elevati rispetto a quelli calcolati impiegando Microwiper.

Questo porta a concludere che l'utilizzo delle strategie proposte da Du et al. conducono ad una diminuzione significativa dei parametri fondamentali per la migrazione *live*: in particolare, risulta del tutto conveniente ordinare le pagine della memoria per *dirtying-rate* crescente, e non trasmettere quelle con *rate* elevata fino alla fase di *stop-and-copy*.

Capitolo 11

Conclusioni

Il lavoro svolto in questa tesi s'inquadra nell'ambito delle problematiche di rete relative alle applicazioni di *Cloud Computing*.
Alla base del modello *cloud* ci sono la *virtualizzazione* e la *migrazione in tempo reale*, tecnologie sulle quali si è, quindi, deciso di concentrare l'attenzione.
La *virtualizzazione* è un ingrediente chiave perché permette l'utilizzo delle risorse hardware in maniera controllata e senza sprechi, consente la condivisione di queste mantenendo l'isolamento dei software che eseguono, conduce all'indipendenza dall'hardware, e permette, quindi, la migrazione delle macchine virtuali.
La *migrazione live* è il trasferimento di una o piú macchine, comprensive di memoria, rete e disco, da un host fisico ad un altro, mentre continua l'esecuzione dei servizi che offrono. Il trasferimento dell'elemento che maggiormente comporta il consumo di tempo e risorse è quello della memoria, ma è assolutamente necessario prestare attenzione anche alla connettivitá di rete e al disco.
Per quel che riguarda proprio il trasferimento della memoria si sono presentati due approcci: quello *pre-copy* e quello *post-copy*. Per approfondire lo studio della migrazione si è peró deciso di concentrarsi sul quello *pre-copy* in quanto è lo standard de facto della migrazione di macchine virtuali: è l'approccio utilizzato dagli hypervisor piú diffusi.
Nella costruzione di un modello valido che descrivesse il processo di migrazione in tempo reale di una VM, si è considerata costante la *dirtying-rate*, per qualsiasi pagina e qualsiasi workload, nonostante questa dipenda, invece, fortemente dal tipo di applicazione in esecuzione; si è fissato il rate di trasmissione, e la dimensione della memoria della macchina che s'intende migrare.

A questo punto, si sono proposti un modello matematico ed uno simulativo in grado di descrivere il sistema in esame: mediante diverse comparazioni abbiamo verificato che entrambi portavano agli stessi risultati, quindi abbiamo impiegato il simulatore per calcolare i valori degli indici di prestazione che interessano la *migrazione live*.
Con riferimento al trasferimento di una singola macchina virtuale alla volta (richieste di migrazione successive vengono messe *in coda*), abbiamo dimostrato come il *tempo medio di migrazione*, tempo durante il quale si tengono occupate le risorse, e il *downtime*, tempo durante il quale si verifica l'interruzione del servizio, dipendano dalla dimensione della memoria, dalla soglia di dati che si vuole trasferire durante la *stop-and-copy* e, soprattutto, dal rate di trasmissione e dalla *dirtying-rate*: in particolare, abbiamo osservato come i valori di entrambi gli indici decrescessero all'aumentare del rate di trasmissione delle pagine, come il valore del *tempo di migrazione* diminuisse all'aumentare della *dirtying-rate*, mentre il *downtime* calava. Abbiamo anche dimostrato come ció fosse direttamente conseguente alle definizioni date per i due indici.
Proseguendo, abbiamo esteso lo studio della migrazione alle *reti* di macchine virtuali: abbiamo, prima di tutto, determinato quali processi si potevano considerare, dopodiché abbiamo analizzato i rispettivi indici di prestazione.
Per quanto riguarda i processi, abbiamo valutato la migrazione di M macchine virtuali identiche *in serie*, e *in parallelo*: si è ipotizzato, in entrambi i casi, che la migrazione poteva ritenersi conclusa solo quando tutte le macchine di un *insieme* erano state trasferite da un sito all'altro.
Anche in queste situazioni abbiamo validato i risultati dei simulatori attraverso i confronti con quelli ottenuti sfruttando gli approcci matematici proposti. A tal proposito, abbiamo osservato come, a paritá dei valori dei parametri di progetto, il *tempo di migrazione* del processo *in serie* fosse minore di quello del processo *in parallelo*. Questo, perché il numero d'iterazioni di *pre-copy* che interessa la migrazione *in parallelo* è superiore a quello che interessa la migrazione *in serie* in quanto, nel primo processo, le macchine virtuali devono dividersi equamente (per ipotesi) la banda. Ci siamo concentrati anche sul confronto del *downtime*: abbiamo notato come i valori di quest'indice siano notevolmente minori nel caso di migrazione *in parallelo*. Infatti, essendo tutte le M macchine di un *set* identiche tra loro, il tempo di *downtime* dell'*insieme* è il tempo di *downtime* di una singola macchina (ottenuto considerando il rate di trasmissione ridotto di un fattore M). Non è lo stesso per la migrazione *in serie*: per questo processo, la definizione corretta di *downtime* è la differenza

tra il tempo totale di migrazione e l'istante in cui inizia la fase di *stop-and-copy* della prima macchina del *set*.
Ritenuta piú efficiente la migrazione *live* in parallelo vista la breve interruzione di servizio che comportava, si è deciso di approfondire lo studio considerando la possibilitá che l'insieme delle M macchine non comprendesse VM identiche. Analizzando attentamente anche questo aspetto, si è notato come gli indici di prestazione dipendessero fortemente dalla differenza tra le dimensioni delle memorie delle macchine coinvolte.
Si è dunque studiato l'andamento del *tempo di migrazione* e del *downtime* in funzione della *varianza* della dimensione della memoria: all'aumentare del valore di $var(V)$ aumenta notevolmente il *downtime*, mentre diminuisce il *tempo di migrazione*. Questo perché si fa, appunto, riferimento al processo *in parallelo*, che prevede la condivisione della banda tra le M macchine: piú le memorie delle VM sono simili in dimensione, piú lungo sará il tempo in cui condivideranno la banda con conseguente diminuzione del rate di trasmissione di un pagina; piú le dimensioni sono diverse, prima si libera la banda per la trasmissione delle pagine delle altre VM ad un *rate* piú elevato. Per quanto riguarda l'aumento del *downtime*, è conseguente alla sua definizione, ovvero alla crescente differenza tra l'istante in cui termina la migrazione e l'istante in cui la prima macchina inizia la fase di *stop-and-copy* nel caso di macchine aventi dimensioni delle memorie sempre piú differenti.

Da tutte le analisi e le comparazioni effettuate, abbiamo stabilito come la scelta del processo da impiegare nella migrazione di reti di macchine virtuali dipenda dal servizio che si fornisce e dalle risorse a disposizione. A tal proposito, abbiamo determinato la convenienza della migrazione in parallelo nel caso si voglia limitare il tempo di disservizio e, sempre in questo contesto, abbiamo verificato come convenga, se possibile, raggruppare in un *insieme* macchine aventi dimensioni delle memorie simili.

Dunque, grazie alla costruzione del modello e una volta note le dimensioni delle memorie, il valore della soglia di dati trasmissibili durante la *stop-and-copy* e la *dirtying-rate* delle pagine di memoria delle VM coinvolte, si potrá dimensionare la banda da garantire alla migrazione che permetta di rispettare determinati vincoli sul *tempo medio di migrazione*, e sul *downtime*.

Appendice A

Codice del simulatore *ad eventi* per la migrazione di macchine virtuali

In questa appendice è riportato il codice del *simulatore ad eventi* impiegato nella simulazione della migrazione di macchine virtuali, aventi tutte la stessa dimensione della memoria.
Questo codice considera il caso in cui la migrazione di una VM possa avvenire solo al termine dell'avvenuta migrazione della VM precedente.
Si ricorda che tale *simulatore* accetta in ingresso almeno cinque valori, rispettivamente per:

- $seed$, che indica il "seme";
- $N - samples$, che indica il numero di eventi che s'intende valutare;
- $N - classes$, che indica il numero di classi che si vogliono considerare;
- $rho - class$, che indica la percentuale di tempo in cui il servitore è impegnato a servire la coda;
- $serv - class$, che indica per quanto tempo l'utente occupa mediamente il servitore.

Sono inoltre fissati il rate di trasmissione delle pagine di memoria: $R = 1$, e il rate con cui si modificano tali pagine: $D00.5$.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Define the event types
#define MIGRATION_REQ 1
#define ITERATION 2
#define END_MIGRATION 3

// Define the maximum number of traffic classes
 allowed by the simulator
#define MAX_CLASSES 10

// Define whether or not to print debug messages
#define DEBUG 0

#define R 1 //rate di trasmissione della
memoria
#define D 0.5 //rate con cui si
modificano le pagine di memoria

// Structure representing an event
struct event {
double time;     // Event occurrence time
char type;       // Event type
char pclass;     // Class of the customer
involved in the event
double mem_size;     // Memory size
int num_iteration;     // Number of iteration
double time_req;     //Request occurence time
double time_start_mig; //Start migration time
double downtime;     //downtime
struct event *next;     // Pointer to the next
event in the list
};
```

```
// Structure representing a customer
waiting in the queue
struct customer {
double t_arr;    // Time of arrival
double mem_size;  // Service time
struct customer *next;  // Pointer to
the next customer in the list
};
// Event list
struct event *event_list;

// List of waiting customers
struct customer *q[MAX_CLASSES];

// Global variables
int seed; // Random seed
int N;    // Number of arrivals to be simulated
int C;    // Number of traffic classes
int k;    // State of the system = number of
customers in the system
int tot_arrivals, tot_departures;
int arrivals[MAX_CLASSES];
double now;
double tot_w, tot_mig, tot_downtime, tot_iter;
double w[MAX_CLASSES], Tdw[MAX_CLASSES],
Tmig[MAX_CLASSES], iter_num[MAX_CLASSES];
double lambda[MAX_CLASSES], mu[MAX_CLASSES];

// This function inserts a new event in the event list
ordered by time
// Arguments:
//  time = occurrence time of the new event
//  type = type of the new event
//  pclass = class of the customer involved in the new event
//  service = service time of the arriving customer, in case the
 event is an arrival, 0 otherwise
void insert_new_event(double time, char type, char pclass,
```

```
double mem_size, int num_iteration, double time_req,
double time_start_mig, double downtime) {

// Declare some useful pointers
struct event *w1,*w2,*w3;
// Create the new event using the provided arguments
w3 = (struct event *) malloc(sizeof(struct event));
w3->time = time;
w3->type = type;
w3->pclass = pclass;
w3->mem_size = mem_size;
w3->num_iteration = num_iteration;
w3->time_req= time_req;
w3->time_start_mig = time_start_mig;
w3->downtime = downtime;

if (DEBUG) fprintf(stderr,"DEBUG 12: New event object
created: %d n",w3);

// Insert the new event in the event list keeping the
correct order
if(event_list == NULL) {  // The event list is empty:
set it to point to the new event
w3->next = NULL;
event_list = w3;
if (DEBUG) fprintf(stderr,"DEBUG 13: Event list was
empty, new event added to list
head: %d n",event_list);
}
else if (event_list->time > w3->time) {
w3->next = event_list;
event_list = w3;
if (DEBUG) fprintf(stderr,"DEBUG 14: Event list
was not empty, new event added to list head:
%d n",event_list);
}
else {  // In all the other cases, move pointers w1 and
```

```
 w2 along the list until the correct point is found,
// then insert the new event
w1 = event_list;
w2 = event_list->next;
while ((w2 != NULL) && (w2->time <= w3->time)) {
w1 = w2;
w2 = w2->next;
}
w1->next = w3;
w3->next = w2;
if (DEBUG) fprintf(stderr,"DEBUG 15: Event list was
not empty, new event added to list n",event_list);
}

if (DEBUG) fprintf(stderr,"DEBUG 01: New event inserted:
time = \%1.6f, type = \%s, class = \%d, mem_size = \%1.6f,
num_iteration = \%d, time_req = \%1.6f,
time_start_mig = \%1.6f, downtime = \%1.6f, next = \%d n",
w3->time, (w3->type == 1 ? "MIGRATION_REQ" :
(w3->type == 2 ? "ITERATION" : "END_MIGRATION")),
w3->pclass, w3->mem_size, w3->num_iteration,
w3->time_req,  w3->time_start_mig, w3->downtime,
w3->next);
}

// This function extracts the first event from the event list
struct event * get_event() {

// Declare some useful pointers
struct event *w3;

// The list is empty and it should not be: generate an error
if (event_list == NULL) {
fprintf(stderr,"ERROR: event list is empty when it
should not be n");
exit(-1);
}
```

```
// Return the first event from the list and update the
list pointer
w3 = event_list;
event_list = w3->next;

if (DEBUG) fprintf(stderr,"DEBUG 02: Next event extracted:
time = \%1.6f, type = \%s, class =\%d, mem_size = \%1.6f,
num_iteration = \%d, time_req = \%1.6f,
time_start_mig = \%1.6f, downtime = \%1.6f,
next = \%d n", w3->time, (w3->type == 1 ?
 "MIGRATION_REQ" :
(w3->type == 2 ? "ITERATION" : "END_MIGRATION")),
w3->pclass,  w3->mem_size, w3->num_iteration,
w3->time_req,  w3->time_start_mig, w3->downtime,
w3->next);

return w3;

}
// This function appends a new customer to the list of waiting
customers according to the class
// Arguments:
//  pclass = class of the customer
//  time = arrival time of the customer
//  service = service time of the customer
void append(char pclass, double time, double mem_size) {

// Declare some useful pointers
struct customer *w1,*w2,*w3;

// Create the customer using the provided arguments
w3 = (struct customer *) malloc(sizeof(struct customer));
w3->t_arr = time;
w3->mem_size = mem_size;

// Append the customer to the relevant class waiting list
if(q[pclass] == NULL) {  // The list is empty
```

```
w3->next = NULL;
q[pclass] = w3;
}
else {  // Append it to the end of the list
w1 = q[pclass];
w2 = q[pclass]->next;
while (w2 != NULL) {
w1 = w2;
w2 = w2->next;
}
w1->next = w3;
w3->next = w2;
}

if (DEBUG) fprintf(stderr,"DEBUG 03: New customer added
 to queue \%d: time = \%1.6f, service = \%1.6f, next = \%d",
 pclass, w3->t_arr,w3->mem_size,w3->next);

}

// This function extracts the first customer
from the customer list
// Arguments:
//  pclass = class of the customer
struct customer * get_customer(char pclass) {

// Declare some useful pointers
struct customer *w3;

// The list is empty and it should not be:
generate an error
if (q[pclass] == NULL) {
fprintf(stderr,"ERROR: customer list of class \%d is
empty when it should not be n",pclass);
exit(-1);
}
```

```
// Return the first customer from the list
and update the list pointer
w3 = q[pclass];
q[pclass] = w3->next;
if (DEBUG) fprintf(stderr,"DEBUG 04: Next customer
extracted from queue \%d: time = \%1.6f,
service = \%1.6f, next = \%dn",pclass,
  w3->t_arr,w3->mem_size,w3->next);

return w3;
}
// This function generates an instance of an exponential
random variable
// Arguments:
// param = parameter of the exponential distribution,
i.e., inverse of the mean value
double expon(double param) {

// Declare some useful variables
int rnd_num;
double unif,val;

// Generate a uniform random number between 0 and 1
rnd_num = rand();
if (rnd_num == RAND_MAX) rnd_num--;
unif = (double)rnd_num/RAND_MAX;

// Transform the uniform random variable into an
 exponential one using the inverse function rule

val = -log(1.0-unif)/param;

if (DEBUG) fprintf(stderr,"DEBUG 05: New exponential
random variable with parameter \%1.6f:
generated value = \%1.6f n",param,val);
return val;
}
```

```
// This function generates an instance of the service
time random variable
// Arguments:
// pclass = class of the customer
double serv(char pclass) {
// Returns an exponential service time
return expon(mu[pclass]);
}

// Initialization function
void initialize() {

int j;

// Initialize all the global variables and counters
k = 0; tfree = 0.0; now = 0.0;
tot_arrivals = 0; tot_departures = 0; tot_w = 0.0; tot_mig = 0.0;
tot_downtime = 0.0;
for (j = 0; j < C; j++) {
arrivals[j] = 0; w[j] = 0.0; Tdw[j] = 0.0; Tmig[j] = 0.0;
// Initialize the list of events by inserting the first arrival
for each class
insert_new_event(expon(lambda[j]),MIGRATION_REQ,j,
1, 0,0,0,0);
q[j]=NULL;
}
}

// Main program
int main(int argc, char* argv[]) {

// Define some useful variables and pointers
int j;
int x=1;
double V,transfertime, dt;
double Vth = 0.1;
```

```
char typ;
struct event *e, *aux;
struct customer *c, *caux;

// Check the command line arguments: must be at least 6,
including the simulator executable
// Otherwise generate an error
if (argc < 6) {
fprintf(stderr,"Usage: \%s <seed> <$N_samples$>
<$N_classes$> <$rho_class$0> <$serv_class$0>
<$rho_class$1> <$serv_class$1> ...",argv[0]);
exit(-1);
}
// Get the random generator seed from the first argument
seed = atoi(argv[1]);
// If the provided seed is zero, use the current timestamp
if (seed == 0) seed = time(NULL);
srand(seed);
// Get the number of customers to be simulated from the
second argument
N = atoi(argv[2]);

// Get the number of classes from the third argument
(no more than MAX_CLASSES)
C = atoi(argv[3]);
if (C > MAX_CLASSES) C = MAX_CLASSES;

// Check if there are enough arguments for each class
(load + average service time for each class)
// Otherwise generate an error
if (argc < 2*C+4) {
fprintf(stderr,"Missing load and service time for
classes \%d to \%d n",(argc-4)/2,C-1);
exit(-1);
}
// Get the load and average service time for each class from the
  following arguments and set
```

```
//mu = 1/service and lambda = rho*mu
for (j = 0; j < C; j++) {
mu[j] = 1.0/atof(argv[4+2*j+1]);
lambda[j] = atof(argv[4+2*j])*mu[j];
}

// Initialize the simulator
initialize();

// Start the main loop
while (tot_departures < N) {
if (DEBUG) {  // Print lots of information about the
 lists if DEBUG is true fprintf(stderr,"DEBUG 17: Cycling...n");
fprintf(stderr,"          Current event list:n");
fprintf(stderr,"          ");
aux = event_list;
while (aux != NULL) {
fprintf(stderr,"[ addr = \%d, time = \%1.6f,
next = \%d ]----->",aux,aux->time,aux->next);
aux = aux->next;
}
fprintf(stderr,"n");
}
if (DEBUG) {
fprintf(stderr,"          Current customer list:n");
for (j=0;j<C;j++) {
fprintf(stderr,"          q[\%d]----->",j);
caux = q[j];
while (caux != NULL) {
fprintf(stderr,"[ addr = \%d, time = \%1.6f,
next = \%d ]----->",caux,caux->t_arr,
caux->next);
caux = caux->next;
}
fprintf(stderr,"n");
}
}
```

```
// Start processing the first event in the list and set
 the current time
e = get_event();
now = e->time;
if (DEBUG) fprintf(stderr,"DEBUG 10:
Current time = \%1.6f n",now);

if (e->type == MIGRATION_REQ) {
k++; tot_arrivals++; arrivals[e->pclass]++;
if (DEBUG) fprintf(stderr,"DEBUG 06: Arrival: time = \%1.6f,
class = \%d, mem_size = \%1.6f n",e->time,e->pclass,
e->mem_size);
V = e->mem_size;
             //printf(" n# Memory Size \%f  n",V);
transfertime = V/R;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
}
else {
typ =ITERATION;
dt = 0.0;
}
if (k == 1) {  // The system was empty: the customer
goes directly into service
// and a departure event is created and inserted into the list
//tfree = now + e->service;
insert_new_event(now + transfertime, typ, e->pclass,
D*transfertime,1, e->time, e->time, dt);
}
else {  // The system was not empty: the customer
is queued to the relevant waiting list
//tfree = tfree + e->service;
append(e->pclass, e->time, e->mem_size);
}
if (tot_arrivals < N) {
insert_new_event(now + expon(lambda[e->pclass]),
```

```
MIGRATION_REQ, e->pclass,1, 0, 0, 0, 0);
}
}
else if (e->type == ITERATION) { // The event is an iteration
//x=x+1;
if (DEBUG) fprintf(stderr,"DEBUG 18: Iteration: time = \%1.6f,
 class = \%d, mem_size = \%1.6f,
 num_iteration = \%d n",e->time,e->pclass,
e->mem_size,e->num_iteration);
V = e->mem_size;
transfertime = V/R;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
}
else {
typ =ITERATION;
dt = 0.0;
}
insert_new_event(now + transfertime, typ, e->pclass,
D*transfertime, e->num_iteration + 1, e->time_req,
 e->time_start_mig, dt);
x=x+1;
}
else if (e->type == END_MIGRATION) {
k--; tot_departures++;
if (DEBUG) fprintf(stderr,"DEBUG 16: Departure:
time = \%1.6f, class = \%d, service = \%1.6f n",
e->time,e->pclass,e->mem_size);
Tdw[e->pclass] = Tdw[e->pclass] + e->downtime;
Tmig[e->pclass] = Tmig[e->pclass] + (now - e->time_start_mig);
iter_num[e->pclass] = iter_num[e->pclass] + e->num_iteration;
if (k > 0) {
j = 0;
while (q[j] == NULL) j++;
c = get_customer(j);
w[j] = w[j] + (now - c->t_arr);  j
```

```
V = c->mem_size;
transfertime = V/R;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
}

else {
typ =ITERATION;
dt = 0.0;
}
insert_new_event(now + transfertime, typ, j, D*transfertime, 1,
c->t_arr, now, dt);
if (DEBUG) fprintf(stderr,"DEBUG 07:
Removing customer object n");
if (c == NULL) {
fprintf(stderr,"ERROR: customer object is NULL when it
should not be n");
exit(-1);
}
free(c);  // Remove the customer object and free memory space
}
}
if (DEBUG) fprintf(stderr,"DEBUG 08: Removing event object n");
if (e == NULL) {
fprintf(stderr,"ERROR: event object is NULL when it should
not be n");
exit(-1);
}
free(e);  // Remove the event object and free memory space
if (DEBUG) fprintf(stderr,"DEBUG 09:
Current state = \%d n",k);
if (DEBUG) fprintf(stderr,"DEBUG 11: Total arrivals = \%d,
Total departures = \%d n",
tot_arrivals,tot_departures);
if (DEBUG) fprintf(stderr,"----------------------
----------------------
```

```
--------------------
------------------ n");

}  // End of the main loop

// Update and print some variables
for (j = 0; j < C; j++) {
tot_w = tot_w + w[j];
tot_mig = tot_mig + Tmig[j];
tot_downtime = tot_downtime + Tdw[j];
tot_iter = tot_iter + iter_num[j];
w[j] = w[j]/arrivals[j];
Tmig[j] = Tmig[j]/arrivals[j];
Tdw[j] = Tdw[j]/arrivals[j];
iter_num[j] = iter_num[j]/arrivals[j];
}
tot_w = tot_w/N;
tot_downtime = tot_downtime / N;
tot_mig = tot_mig / N;
tot_iter = tot_iter / N;

for (j = 0; j < C; j++) {
printf(" n# Class t tMean waiting time tMean migration time
tMean downtime tMean number of iterations n");
printf("  \%d t t\%1.6f t t \%1.6f t t \%1.6f t t \%1.6f n",j,w[j],
Tmig[j],Tdw[j],iter_num[j]);
}
printf(" n# x \%d  n",x);
printf(" n# Overall: tMean waiting time tMean migration time
tMean downtime tMean number of iterations n");
printf(" t t \%1.6f t t \%1.6f t t \%1.6f t t \%1.6f n",
tot_w,tot_mig,tot_downtime,tot_iter);
    }
```

Appendice B

Codice del simulatore per la migrazione *in serie* di macchine virtuali

Per la simulazione della migrazione di M macchine virtuali in serie si è fatto riferimento al codice di seguito proposto.
Si sono definiti e fissati il rate di trasmissione delle pagine e il rate con cui queste si modificano:

- $R = 1$
- $D = 0.2$

Anche per questo *simulatore ad eventi* è necessario specificare un valore per ciascuno dei parametri d'ingresso (si ricorda che sono almeno cinque):

- $seed = 1$
- $N - samples = 4$
- $N - classes = 1$
- $rho - class = 0.1$
- $serv - class = 1$

Il codice per *simulare la migrazione di M macchine virtuali in serie* è il seguente.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Define the event types
#define MIGRATION_REQ 1
#define ITERATION 2
#define END_MIGRATION 3

// Define the maximum number of
traffic classes allowed
by the simulator
#define MAX_CLASSES 10

// Define whether or not to
print debug messages
#define DEBUG 0

#define R 1
#define D 0.2

// Structure representing an event
struct event {
double time;     // Event occurrence time
char type;       // Event type
char pclass;     // Class of the
customer involved in the event
double mem_size;     // Memory size
int num_iteration;     // Number of iteration
double time_req;     //Request occurence time
double time_start_mig; //Start migration time
double downtime;     //downtime
struct event *next;     // Pointer to the
next event in the list
};
```

```
// Structure representing a customer
waiting in the queue
struct customer {
double t_arr;    // Time of arrival
double mem_size;  // Service time
struct customer *next;  // Pointer
to the next customer
in the list
};

// Event list
struct event *event_list;

// List of waiting customers
struct customer *q[MAX_CLASSES];

// Global variables
int seed; // Random seed
int N;    // Number of arrivals to be simulated
int M = 2; // numero di VM in un insieme
int C;    // Number of traffic classes
int k;    // State of the system
int tot_arrivals, tot_departures; // Counters for the
total number of arrivals/departures
int arrivals[MAX_CLASSES]; // Array of
counters for the number of
arrivals of each class
int Nset = 0;
double tfree; // Time when the server
has become or will become idle
double now;   // Current time
double tot_w, tot_mig, tot_downtime, tot_iter;
double w[MAX_CLASSES], Tmigra[MAX_CLASSES];
double Tmigration[MAX_CLASSES], Tdown[MAX_CLASSES];
double Tdw[MAX_CLASSES], Tm[MAX_CLASSES];
double Tmig[MAX_CLASSES], iter_num[MAX_CLASSES];
double lambda[MAX_CLASSES], mu[MAX_CLASSES];
```

```
double load[MAX_CLASSES]; // carico effettivo del sistema
//double Vmem=2;

// This function inserts a new event in the
event list ordered by time
// Arguments:
//  time = occurrence time of the new event
//  type = type of the new event
//  pclass = class of the customer
involved in the new event
//  service = service time of the arriving
customer, in case the event
is an arrival, 0 otherwise
void insert_new_event(double time,
char type, char pclass,
double mem_size,
int num_iteration,
double time_req,
double time_start_mig,
double downtime) {

// Declare some useful pointers
struct event *w1,*w2,*w3;

// Create the new event
using the provided arguments
w3 = (struct event *) malloc(sizeof(struct event));
w3->time = time;
w3->type = type;
w3->pclass = pclass;
w3->mem_size = mem_size;
w3->num_iteration = num_iteration;
w3->time_req= time_req;
w3->time_start_mig = time_start_mig;
w3->downtime = downtime;

if (DEBUG) fprintf(stderr,"DEBUG 12:
```

```
New event object created: %d\n",w3);

// Insert the new event in the
event list keeping the correct order
if(event_list == NULL) {  // The event
list is empty: set it to
point to the new event
w3->next = NULL;
event_list = w3;
if (DEBUG) fprintf(stderr,"DEBUG 13:
Event list was empty, new event
added to list head: %d\n",event_list);
}
else if (event_list->time > w3->time) {
w3->next = event_list;
event_list = w3;
if (DEBUG) fprintf(stderr,"DEBUG 14:
Event list was not empty,
new event added to list head:
%d\n",event_list);
}
else {
w1 = event_list;
w2 = event_list->next;
while ((w2 != NULL)
&& (w2->time <= w3->time)) {
w1 = w2;
w2 = w2->next;
}
w1->next = w3;
w3->next = w2;
if (DEBUG) fprintf(stderr,"DEBUG 15:
Event list was not empty,
new event added to list\n",
event_list);
}
```

```
if (DEBUG) fprintf(stderr,"DEBUG 01:
New event inserted: time = %1.6f,
type = %s,
class = %d,
mem_size = %1.6f,
num_iteration = %d,
time_req = %1.6f,
time_start_mig = %1.6f,
downtime = %1.6f,
next = %d\n", w3->time,
(w3->type == 1 ? "MIGRATION_REQ"
: (w3->type == 2 ? "ITERATION"
: "END_MIGRATION")),
w3->pclass, w3->mem_size,
w3->num_iteration, w3->time_req,
w3->time_start_mig, w3->downtime,
w3->next);

}

// This function extracts the first
event from the event list
struct event * get_event() {

// Declare some useful pointers
struct event *w3;

// The list is empty and it should
not be: generate an error
if (event_list == NULL) {
fprintf(stderr,"ERROR:
event list is empty
when it should not be\n");
exit(-1);
}

// Return the first event from the
```

```
list and update the list pointer
w3 = event_list;
event_list = w3->next;

if (DEBUG) fprintf(stderr,"DEBUG 02:
Next event extracted: time = %1.6f,
type = %s,
class = %d,
mem_size = %1.6f,
num_iteration = %d,
time_req = %1.6f,
time_start_mig = %1.6f,
downtime = %1.6f, next = %d\n",
w3->time, (w3->type == 1 ?
"MIGRATION_REQ" :
(w3->type == 2 ?
"ITERATION" :
"END_MIGRATION")),
w3->pclass, w3->mem_size,
w3->num_iteration, w3->time_req,
w3->time_start_mig, w3->downtime,
w3->next);

return w3;

}

// This function appends a new
customer to the list of waiting
customers according to the class
// Arguments:
//  pclass = class of the customer
//  time = arrival time of the customer
//  service = service time of the customer
void append(char pclass, double time,
double mem_size) {
```

```
// Declare some useful pointers
struct customer *w1,*w2,*w3;

// Create the customer using the
provided arguments
w3 = (struct customer *) malloc(sizeof(struct customer));
w3->t_arr = time;
w3->mem_size = mem_size;

// Append the customer to
the relevant class waiting list
if(q[pclass] == NULL) {  // The list is empty
w3->next = NULL;
q[pclass] = w3;
}
else {  // Append it to the end of the list
w1 = q[pclass];
w2 = q[pclass]->next;
while (w2 != NULL) {
w1 = w2;
w2 = w2->next;
}
w1->next = w3;
w3->next = w2;
}

if (DEBUG) fprintf(stderr,"DEBUG 03:
New customer added to queue %d: t
ime = %1.6f,
service = %1.6f,
next = %d\n",
pclass,w3->t_arr,
w3->mem_size,
w3->next);

}
```

```
// This function extracts the first
customer from the customer list
// Arguments:
//  pclass = class of the customer
struct customer * get_customer(char pclass) {

// Declare some useful pointers
struct customer *w3;

// The list is empty and it should not be:
generate an error
if (q[pclass] == NULL) {
fprintf(stderr,"ERROR:
customer list of class
%d is empty when it
should not be\n",pclass);
exit(-1);
}

// Return the first customer from the
list and update the list pointer
w3 = q[pclass];
q[pclass] = w3->next;

if (DEBUG) fprintf(stderr,"DEBUG 04:
Next customer extracted from queue %d:
time = %1.6f,
service = %1.6f,
next = %d\n",
pclass,w3->t_arr,w3->mem_size,
w3->next);

return w3;

}

// This function generates an instance
```

```
of an exponential random variable
// Arguments:
//  param = parameter of the exponential
distribution, i.e., inverse
of the mean value
double expon(double param) {

// Declare some useful variables
int rnd_num;
double unif,val;

// Generate a uniform random
number between 0 and 1
rnd_num = rand();
if (rnd_num == RAND_MAX) rnd_num--;
unif = (double)rnd_num/RAND_MAX;

// Transform the uniform random variable
 into an exponential one using
 the inverse function rule
val = -log(1.0-unif)/param;
if (DEBUG) fprintf(stderr,"DEBUG 05:
New exponential random
variable with parameter %1.6f:
generated value = %1.6f\n",
param,val);

return val;

}

// This function generates an instance
of the service time random variable
// Arguments:
//  pclass = class of the customer
double serv(char pclass) {
```

```
// Returns an exponential service time
return expon(mu[pclass]);

}

// Initialization function
void initialize() {

int j;

// Initialize all the global variables and counters
k = 0; tfree = 0.0; now = 0.0;
tot_arrivals = 0; tot_departures = 0;
 tot_w = 0.0; tot_mig = 0.0;
 tot_downtime = 0.0;
for (j = 0; j < C; j++) {
arrivals[j] = 0; w[j] = 0.0;
Tdw[j] = 0.0; Tmig[j] = 0.0;
// Initialize the list of events by
inserting the first arrival for each class

insert_new_event(expon(lambda[j]),
MIGRATION_REQ,j,serv(j), 0,0,0,0);
q[j]=NULL;
}

}

// Main program
int main(int argc, char* argv[]) {

// Define some useful variables and pointers
int j;
int x=1;
    int a = 1;
double V,transfertime, dt;
double Vth = 0.1;
```

```
char typ;
struct event *e, *aux;
struct customer *c, *caux;

// Check the command line arguments:
must be at least 5, including the
simulator executable
// Otherwise generate an error
if (argc < 6) {
fprintf(stderr,"Usage: %s <seed>
<N_samples> <N_classes>
<rho_class0> <serv_class0>
<rho_class1> <serv_class1> ...\n",
argv[0]);
exit(-1);
}

// Get the random generator seed f
rom the first argument
seed = atoi(argv[1]);
// If the provided seed is zero,
use the current timestamp
if (seed == 0) seed = time(NULL);
srand(seed);

// Get the number of customers
to be simulated from the
second argument
N = atoi(argv[2]);

// Get the number of classes from the
third argument (no more than
MAX_CLASSES)
C = atoi(argv[3]);
if (C > MAX_CLASSES) C = MAX_CLASSES;

// Check if there are enough
```

```
arguments for each class
(load + average service time
 for each class)
// Otherwise generate an error
if (argc < 2*C+4) {
fprintf(stderr,"Missing load and
service time for classes
%d to %d\n",(argc-4)/2,C-1);
exit(-1);
}
// Get the load and average service
time for each class from the
following arguments
// and set mu = 1/service and lambda = rho*mu
for (j = 0; j < C; j++) {
mu[j] = 1.0/atof(argv[4+2*j+1]);
lambda[j] = atof(argv[4+2*j])*mu[j];
}

// Initialize the simulator
initialize();

// Start the main loop
while (tot_departures < N) {

if (DEBUG) {  // Print lots of information
about the lists if DEBUG is true
fprintf(stderr,"DEBUG 17: Cycling...\n");
fprintf(stderr,"          Current event list:\n");
fprintf(stderr,"          ");
aux = event_list;
while (aux != NULL) {
fprintf(stderr,"[ addr = %d,
time = %1.6f, next = %d
]----->",aux,aux->time,
aux->next);
aux = aux->next;
```

```
}
fprintf(stderr,"\n");
}
if (DEBUG) {
fprintf(stderr,"   Current customer list:\n");
for (j=0;j<C;j++) {
fprintf(stderr,"          q[%d]----->",j);
caux = q[j];
while (caux != NULL) {
fprintf(stderr,"[ addr = %d,
time = %1.6f, next = %d
 ]----->",caux,caux->t_arr,
 caux->next);
caux = caux->next;
}
fprintf(stderr,"\n");
}
}

// Start processing the first event
in the list and set the current time
e = get_event();
now = e->time;
if (DEBUG) fprintf(stderr,"DEBUG 10:
Current time = %1.6f\n",now);

if (e->type == MIGRATION_REQ) {
k++; tot_arrivals++; arrivals[e->pclass]++;
if (DEBUG) fprintf(stderr,"DEBUG 06:
Arrival: time = %1.6f, class = %d,
mem_size = %1.6f\n",
e->time,e->pclass,e->mem_size);
V = e->mem_size;
transfertime = V/R;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
```

```
}
else {
typ =ITERATION;
dt = 0.0;
}
if (k == 1) {
insert_new_event(now +
transfertime, typ,
e->pclass, D*transfertime,
1, e->time, e->time, dt);
}
else {
append(e->pclass, e->time,
e->mem_size);
}
if (tot_arrivals < N) {
    insert_new_event(now +
      expon(lambda[e->pclass]),
 MIGRATION_REQ,e->pclass,
 serv(e->pclass), 0, 0, 0, 0);
}
}
else if (e->type == ITERATION) {
if (DEBUG) fprintf(stderr,"DEBUG 18:
Iteration: time = %1.6f,
class = %d, mem_size = %1.6f,
num_iteration = %d\n",e->time,
e->pclass,e->mem_size,
e->num_iteration);
V = e->mem_size;
transfertime = V/R;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
}
else {
typ =ITERATION;
```

```
dt = 0.0;
}
insert_new_event(now +
transfertime, typ,
e->pclass, D*transfertime,
e->num_iteration + 1,
e->time_req,
e->time_start_mig, dt);
x=x+1;
}
else if (e->type == END_MIGRATION) {
k--; tot_departures++;
if (DEBUG) fprintf(stderr,"DEBUG 16:
Departure: time = %1.6f,
class = %d,
service = %1.6f\n",
e->time,e->pclass,
e->mem_size);
             Tmig[e->pclass] = Tmig[e->pclass] +
(now - e->time_start_mig);
iter_num[e->pclass] = iter_num[e->pclass] +
e->num_iteration -1;
             w[e->pclass] = w[e->pclass] +
(now - e->time_req);
             if ( a == 1) {
                 Tdw[e->pclass] = Tdw[e->pclass] +
e->downtime;
                 Tdown [e->pclass] = Tdown [e->pclass] +
  Tdw[e->pclass];
                 Tmigration[e->pclass]=now -
   e->time_start_mig;
                 Tm[e->pclass] = Tm[e->pclass] +
Tmigration[e->pclass];
                 a++;
             }
             else if (a != 1 && a < M) {
                 a++;
```

```
            }
            else {
                Tmigra[e->pclass]= Tmig[e->pclass] -
Tmigration[e->pclass];
                Tdw[e->pclass] = Tdw[e->pclass] +
 Tmigra[e->pclass];
                Tdown [e->pclass] = Tdown [e->pclass] +
                 Tdw[e->pclass];
                Tm[e->pclass] = Tm[e->pclass] +
                Tmig[e->pclass]-
Tmigration[e->pclass];
                Tmig[e->pclass] = 0;
                a = 1;
                Nset++;
            }
            Tdw[e->pclass] = 0;
if (k > 0) {
j = 0;
while (q[j] == NULL) j++;
c = get_customer(j);
w[j] = w[j] + (now - c->t_arr);
V = c->mem_size;
transfertime = V/R;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
}
else {
typ =ITERATION;
dt = 0.0;
}
insert_new_event(now + transfertime,
typ, j, D*transfertime, 1,
c->t_arr, now, dt);
if (DEBUG) fprintf(stderr,"DEBUG 07:
Removing customer object\n");
if (c == NULL) {
```

```
fprintf(stderr,"ERROR:
customer object is NULL
when it should not be\n");
exit(-1);
}
free(c);  // Remove the customer
object and free
memory space
}
}
if (DEBUG) fprintf(stderr,"DEBUG 08:
Removing event object\n");
if (e == NULL) {
fprintf(stderr,"ERROR: event object
is NULL when it should
not be\n");
exit(-1);
}
free(e);  // Remove the
event object and
free memory space

if (DEBUG) fprintf(stderr,"DEBUG 09:
Current state = %d\n",k);
if (DEBUG) fprintf(stderr,"DEBUG 11:
Total arrivals = %d,
Total departures = %d\n",
tot_arrivals,tot_departures);
if (DEBUG) fprintf(stderr,"-------------
---------------------------
---------------------------
--------------\n\n");

}  // End of the main loop

// Update and print some variables
for (j = 0; j < C; j++) {
```

```
tot_w = tot_w + w[j];
tot_mig = tot_mig + Tm[j];
tot_downtime = tot_downtime +
Tdown[j];
tot_iter = tot_iter + iter_num[j];
w[j] = w[j]/(arrivals[j]/M);
Tm[j] = Tm[j]/(arrivals[j]/M);
         Tdown[j] = Tdown[j]/(arrivals[j]/M);
         load[j] = Tm[j]* lambda[j];
iter_num[j] = iter_num[j]/arrivals[j];
}
tot_w = tot_w/(N/M);
tot_downtime = tot_downtime / (N/M);
tot_mig = tot_mig / (N/M);
tot_iter = tot_iter / (N);

for (j = 0; j < C; j++) {
printf("\n# Class\t\t
Mean waiting time\t
Mean migration time\t
Mean downtime\t
Mean number of iterations\n");
printf("  %d\t\t%1.6f\t\t%1.6f\t\t%1.6f\t\
t%1.6f\n",j,w[j],Tm[j],Tdown[j],iter_num[j]);
        printf("\n# Carico effettivo \%f \n",load[j]);
}
    printf("\n# Nset \%d \n", Nset);
printf("\n# Overall:\t
Mean waiting time\t
Mean migration time\t
Mean downtime\t
Mean number of iterations\n");
printf("\t\t%1.6f\t\t%1.6f\t\t%1.6f\t\t%1.6f\n",
tot_w,tot_mig,tot_downtime,tot_iter);
}
```

Appendice C

Codice del simulatore per la migrazione *in parallelo* di macchine virtuali

In questa appendice è proposto il codice impiegato nella *simulazione di migrazione di M macchine virtuali in parallelo.*
Sono definiti e fissati i valori di R, D e M. Per esempio:

- $R = 1$
- $D = 0.2$
- $M = 4$

ricordando che, affinché abbia senso l'adozione dell'approccio *pre-copy*, deve essere $D \cdot M < R$.

Questo simulatore prende in ingresso almeno cinque valori, per esempio:

-$seed = 1$

-$N - samples = 8$

-$N - classes = 4$

-$rho - class = 0.4$

-$serv - class = 1$

A seguire il codice.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Define the event types
#define MIGRATION_REQ 1
#define ITERATION 2
#define END_MIGRATION 3

// Define the maximum number
of traffic classes allowed by the simulator
#define MAX_CLASSES 10

// Define whether or not to print
debug messages
#define DEBUG 0

#define R 1
#define D 0.2
#define M 4

// Structure representing an event
struct event {
double time;     // Event occurrence time
char type;       // Event type
char pclass;     // Class of the customer
involved in the event
double mem_size;     // Memory size
int num_iteration;     // Number of iteration
double time_req;     //Request occurence time
double time_start_mig; //Start migration time
double downtime;     //downtime
struct event *next;     // Pointer to the next
event in the list
};
```

```
// Structure representing a
customer waiting in the queue
struct customer {
double t_arr;    // Time of arrival
double mem_size;  // Service time
struct customer *next;  // Pointer
to the next
customer in the list
};

// Event list
struct event *event_list;

// List of waiting customers
struct customer *q[MAX_CLASSES];

// Global variables
int seed; // Random seed
int N;    // Number of arrivals
to be simulated
int Nset;
double r; // Number of VM in a set
int C;    // Number of traffic classes
int k;    // State of the system
int tot_arrivals, tot_departures; // Counters for the
total number of arrivals/departures
int arrivals[MAX_CLASSES]; // Array of counters
for the number of arrivals of each class
double L;
double tfree; // Time when the server
 has become or will become idle
double now;   // Current time
double tot_w, tot_mig, tot_downtime; // Overall
mean waiting time, downtime,
migration time and number of iterations
double w[MAX_CLASSES], Tm[MAX_CLASSES],
```

```
double Tdw[MAX_CLASSES], Tmig[MAX_CLASSES];
double iter_num[MAX_CLASSES], tot_iter;
double lambda[MAX_CLASSES], mu[MAX_CLASSES];
double load[MAX_CLASSES]; // carico effettivo del sistema
//double Vmem=1;

// This function inserts a new event
in the event list ordered by time
// Arguments:
//  time = occurrence time of the
new event
//  type = type of the new event
//  pclass = class of the customer
involved in the new event
//  service = service time of the arriving
customer, in case the event
is an arrival, 0 otherwise
void insert_new_event(double time,
char type,
char pclass,
double mem_size,
int num_iteration,
double time_req,
double time_start_mig,
double downtime) {

// Declare some useful pointers
struct event *w1,*w2,*w3;

// Create the new event
using the provided arguments
w3 = (struct event *) malloc(sizeof(struct event));
w3->time = time;
w3->type = type;
w3->pclass = pclass;
w3->mem_size = mem_size;
w3->num_iteration = num_iteration;
```

```
w3->time_req= time_req;
w3->time_start_mig = time_start_mig;
w3->downtime = downtime;

if (DEBUG) fprintf(stderr,"DEBUG 12:
New event object created: %d\n",w3);

// Insert the new event in the event
 list keeping the correct order
if(event_list == NULL) {  // The event list
is empty: set it to point to the new event
w3->next = NULL;
event_list = w3;
if (DEBUG) fprintf(stderr,"DEBUG 13:
Event list was empty, new event
added to list head: %d\n",event_list);
}
else if (event_list->time > w3->time) {
w3->next = event_list;
event_list = w3;
if (DEBUG) fprintf(stderr,"DEBUG 14:
Event list was not empty,
new event added to list
head: %d\n",event_list);
}
else {
w1 = event_list;
w2 = event_list->next;
while ((w2 != NULL)
&& (w2->time <= w3->time)) {
w1 = w2;
w2 = w2->next;
}
w1->next = w3;
w3->next = w2;
if (DEBUG) fprintf(stderr,"DEBUG 15:
Event list was not empty,
```

```
new event added to list\n",
event_list);
}

if (DEBUG) fprintf(stderr,"DEBUG 01:
New event inserted: time = %1.6f,
type = %s, class = %d,
mem_size = %1.6f,
num_iteration = %d,
time_req = %1.6f,
time_start_mig = %1.6f,
downtime = %1.6f,
next = %d\n",
w3->time, (w3->type == 1
? "MIGRATION_REQ" : (w3->type == 2
? "ITERATION" : "END_MIGRATION")),
w3->pclass, w3->mem_size,
w3->num_iteration, w3->time_req,
w3->time_start_mig, w3->downtime,
w3->next);

}

// This function extracts the first
event from the event list
struct event * get_event() {

// Declare some useful pointers
struct event *w3;

// The list is empty and it should not be:
generate an error
if (event_list == NULL) {
fprintf(stderr,"ERROR: event list
is empty when it should
not be\n");
exit(-1);
```

```
}

// Return the first event from the list
and update the list pointer
w3 = event_list;
event_list = w3->next;

if (DEBUG) fprintf(stderr,"DEBUG 02:
Next event extracted: time = %1.6f,
type = %s, class = %d,
mem_size = %1.6f,
num_iteration = %d,
time_req = %1.6f,
time_start_mig = %1.6f,
downtime = %1.6f,
next = %d\n", w3->time,
(w3->type == 1 ? "MIGRATION_REQ" :
(w3->type == 2 ? "ITERATION" :
"END_MIGRATION")), w3->pclass,
w3->mem_size, w3->num_iteration,
w3->time_req, w3->time_start_mig,
w3->downtime, w3->next);

return w3;

}

// This function appends a new customer
to the list of waiting customers
according to the class
// Arguments:
//  pclass = class of the customer
//  time = arrival time of the customer
//  service = service time of the customer
void append(char pclass, double time,
double mem_size) {
```

```
// Declare some useful pointers
struct customer *w1,*w2,*w3;

// Create the customer using
the provided arguments
w3 = (struct customer *) malloc(sizeof(struct customer));
w3->t_arr = time;
w3->mem_size = mem_size;

// Append the customer to the
relevant class waiting list
if(q[pclass] == NULL) {  // The list is empty
w3->next = NULL;
q[pclass] = w3;
}
else {  // Append it to the end of the list
w1 = q[pclass];
w2 = q[pclass]->next;
while (w2 != NULL) {
w1 = w2;
w2 = w2->next;
}
w1->next = w3;
w3->next = w2;
}

if (DEBUG) fprintf(stderr,"DEBUG 03:
New customer added to queue %d:
time = %1.6f, service = %1.6f,
next = %d\n",pclass,w3->t_arr,
w3->mem_size,w3->next);

}

// This function extracts the first
customer from the customer list
// Arguments:
```

```
//  pclass = class of the customer
struct customer * get_customer(char pclass) {

// Declare some useful pointers
struct customer *w3;

// The list is empty and it should not be:
generate an error
if (q[pclass] == NULL) {
fprintf(stderr,"ERROR: customer
list of class %d is empty
when it should not be\n",
pclass);
exit(-1);
}

// Return the first customer from the list
and update the list pointer
w3 = q[pclass];
q[pclass] = w3->next;

if (DEBUG) fprintf(stderr,"DEBUG 04:
Next customer extracted
from queue %d: time = %1.6f,
service = %1.6f, next = %d\n",
pclass,w3->t_arr,w3->mem_size,
w3->next);

return w3;
}

// This function generates an
instance of an exponential
random variable
// Arguments:
//  param = parameter of the
exponential distribution,
```

```
i.e., inverse of the mean value
double expon(double param) {

// Declare some useful variables
int rnd_num;
double unif,val;

// Generate a uniform random
number between 0 and 1
rnd_num = rand();
if (rnd_num == RAND_MAX) rnd_num--;
unif = (double)rnd_num/RAND_MAX;

// Transform the uniform random
variable into an exponential
one using the inverse function rule
val = -log(1.0-unif)/param;
if (DEBUG) fprintf(stderr,"DEBUG 05:
New exponential random
variable with parameter %1.6f:
generated value = %1.6f\n",param,val);

return val;

}

// This function generates an
 instance of the service
 time random variable
// Arguments:
//  pclass = class of the customer
double serv(char pclass) {

// Returns an exponential service time
return expon(mu[pclass]);

}
```

```
// Initialization function
void initialize() {

int j;

// Initialize all the global
variables and counters
k = 0; tfree = 0.0; now = 0.0;
tot_arrivals = 0; tot_departures = 0;
 tot_w = 0.0; tot_mig = 0.0;
 tot_downtime = 0.0;
for (j = 0; j < C; j++) {
arrivals[j] = 0; w[j] = 0.0;
Tdw[j] = 0.0; Tmig[j] = 0.0;
// Initialize the list of events by inserting the
first arrival for each class

insert_new_event(expon(lambda[j]),
MIGRATION_REQ,j,serv(j), 0, 0, 0, 0);
q[j]=NULL;

}

}

// Main program
int main(int argc, char* argv[]) {

// Define some useful
variables and pointers
int j;
double x=0;
double V,transfertime, dt;
double Vth = 0.1;
char typ;
struct event *e, *aux;
```

```
struct customer *c, *caux;
    r=R/M;
    Nset=N/M;
    //double t=0;


// Check the command line arguments:
must be at least 5, including the
simulator executable
// Otherwise generate an error
if (argc < 6) {
fprintf(stderr,"Usage: %s <seed>
<N_samples> <N_classes>
<rho_class0> <serv_class0>
 <rho_class1> <serv_class1>
 ...\n",argv[0]);
exit(-1);
}

// Get the random generator seed
 from the first argument
seed = atoi(argv[1]);
// If the provided seed is zero,
use the current timestamp
if (seed == 0) seed = time(NULL);
srand(seed);

// Get the number of customers
to be simulated from the
second argument
N = atoi(argv[2]);

// Get the number of classes from
 the third argument
 (no more than MAX_CLASSES)
C = atoi(argv[3]);
if (C > MAX_CLASSES) C = MAX_CLASSES;
```

```
// Check if there are enough arguments for
each class (load + average
service time for each class)
// Otherwise generate an error
if (argc < 2*C+4) {
fprintf(stderr,"Missing load and
service time for classes
%d to %d\n",(argc-4)/2,C-1);
exit(-1);
}
// Get the load and average service time
for each class from the
following arguments
// and set mu = 1/service and lambda = rho*mu
for (j = 0; j < C; j++) {
mu[j] = 1.0/atof(argv[4+2*j+1]);
lambda[j] = atof(argv[4+2*j])*mu[j];
}


// Initialize the simulator
initialize();

// Start the main loop
while (tot_departures < N) {  // Loop until all
 the generated customers
 have left the system

if (DEBUG) {
fprintf(stderr,"DEBUG 17: Cycling...\n");
fprintf(stderr,"          Current event list:\n");
fprintf(stderr,"          ");
aux = event_list;
while (aux != NULL) {
fprintf(stderr,"[ addr = %d,
time = %1.6f, next =
```

```
%d ]----->",aux,aux->time,
aux->next);
aux = aux->next;
}
fprintf(stderr,"\n");
}
if (DEBUG) {
fprintf(stderr,"   Current customer list:\n");
for (j=0;j<C;j++) {
fprintf(stderr,"          q[%d]----->",j);
caux = q[j];
while (caux != NULL) {
fprintf(stderr,"[ addr = %d,
time = %1.6f, next = %d
]----->",caux,caux->t_arr,
caux->next);
caux = caux->next;
}
fprintf(stderr,"\n");
}
}

// Start processing the first event
 in the list and set the current time
e = get_event();
now = e->time;
        //printf("\n# now \%f \n",now);
if (DEBUG) fprintf(stderr,"DEBUG 10:
Current time = %1.6f\n",now);

if (e->type == MIGRATION_REQ) {  // The event
is an arrival:
increment the counters
and the state
k++; tot_arrivals++; arrivals[e->pclass]++;
if (DEBUG) fprintf(stderr,"DEBUG 06:
Arrival: time = %1.6f,
```

```
class = %d,
mem_size = %1.6f\n",
e->time,e->pclass,e->mem_size);
V = e->mem_size;
transfertime = V*M;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
}
else {
typ =ITERATION;
dt = 0.0;
}
if (k == 1) {
insert_new_event(now + transfertime,
typ, e->pclass,
D*transfertime,
e->num_iteration + 1,
e->time, e->time, dt);
}
else {
append(e->pclass, e->time, e->mem_size);
}
if (tot_arrivals < N) {
                     insert_new_event(now +
                      expon(lambda[e->pclass]),
MIGRATION_REQ,e->pclass,
serv(e->pclass), 0, 0, 0, 0);
}
}
else if (e->type == ITERATION) {
 if (DEBUG) fprintf(stderr,"DEBUG 18:
  Iteration: time = %1.6f,
class = %d,
mem_size = %1.6f,
num_iteration = %d\n",
e->time,e->pclass,e->mem_size,
```

```
e->num_iteration);
V = e->mem_size;
transfertime = V*M;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
}
else {
typ =ITERATION;
dt = 0.0;
}
insert_new_event(now +
transfertime, typ,
e->pclass, D*transfertime,
e->num_iteration + 1,
e->time_req, e->time_start_mig,
dt);
x=x+1;
}
else if (e->type == END_MIGRATION) {
k--; tot_departures++;
if (DEBUG) fprintf(stderr,"DEBUG 16:
Departure: time = %1.6f,
class = %d,
service = %1.6f\n",
e->time,e->pclass,
e->mem_size);
Tdw[e->pclass] = Tdw[e->pclass] + e->downtime;
Tmig[e->pclass] = Tmig[e->pclass] +
(now - e->time_start_mig);
if (k > 0) {
j = 0;
while (q[j] == NULL) j++;
c = get_customer(j);
                if(k>1){
                    w[j] = w[j] + (now - c->t_arr);
                }
```

```
V = c->mem_size;
transfertime = V*M;
if (V < Vth) {
typ = END_MIGRATION;
dt = transfertime;
}
else {
typ =ITERATION;
dt = 0.0;
}
insert_new_event(now +
transfertime, typ, j,
D*transfertime,
e->num_iteration + 1,
c->t_arr, now, dt);
if (DEBUG) fprintf(stderr,"DEBUG 07:
Removing customer object\n");
if (c == NULL) {
fprintf(stderr,"ERROR:
customer object is
NULL when it should not be\n");
exit(-1);
}
free(c);
  }

}
if (DEBUG) fprintf(stderr,"DEBUG 08:
Removing event object\n");
if (e == NULL) {
fprintf(stderr,"ERROR: event object is
NULL when it should not be\n");
exit(-1);
}
free(e);  // Remove the event object
and free memory space
```

```
if (DEBUG) fprintf(stderr,"DEBUG 09:
Current state = %d\n",k);
if (DEBUG) fprintf(stderr,"DEBUG 11:
Total arrivals = %d,
Total departures = %d\n",
tot_arrivals,tot_departures);
if (DEBUG) fprintf(stderr,"-------------
-------------------------
-------------------------
-----------------\n\n");

}  // End of the main loop

// Update and print some variables
for (j = 0; j < C; j++) {
tot_w = tot_w + w[j];
tot_mig = tot_mig + Tmig[j];
tot_downtime = tot_downtime + Tdw[j];
        tot_iter = tot_iter + x;
w[j] = w[j]/(arrivals[j]);
Tmig[j] = Tmig[j]/(arrivals[j]);
Tdw[j] = Tdw[j]/(arrivals[j]);
        load[j] = Tmig[j]* lambda[j];
         iter_num[j] = x/(arrivals[j]);
}
tot_w = tot_w/(N);
tot_downtime = tot_downtime / (N);
tot_mig = tot_mig /(N);
tot_iter = tot_iter / (N);

for (j = 0; j < C; j++) {
printf("\n# Class\t\t
Mean waiting time\t
Mean migration time\t
Mean downtime\t
Mean number of iterations\n");
printf("  %d\t\t%1.6f\t\t%1.6f\t\t%1.6f\t
```

```
\t%1.6f\n",j,w[j],Tmig[j],Tdw[j],
iter_num[j]);
        printf("\n# Carico effettivo \%f \n",load[j]);
}
printf("\n# Overall:\t
Mean waiting time\t
Mean migration time\t
Mean downtime\t
Mean number of iterations\n");
printf("\t\t%1.6f\t\t%1.6f\t\t%1.6f\t\t%1.6f\n",
tot_w,tot_mig,tot_downtime,tot_iter);

}
```

Appendice D

Codice del simulatore per la migrazione *in parallelo* di macchine aventi diversa dimensione della memoria

In questa appendice è proposto il codice impiegato nella simulazione della migrazione in parallelo di M macchine virtuali aventi differente dimensione della memoria. In particolare, si è definito il numero di macchine virtuali di un set: $M = 10$, si sono mantenuti $R = 1$ e $D = 0.05$, mentre si è opportunamente scelto di generare casualmente la dimensione della memoria della macchine virtuali coinvolte.

```
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

// Define the event types
#define MIGRATION_REQ 1
#define ITERATION 2
#define END_MIGRATION 3

// Define the maximum number of
```

```
traffic classes allowed by the simulator
#define MAX_CLASSES 10

// Define whether or not to print debug messages
#define DEBUG 0

#define R 1
#define D 0.05 // IMPORTANTE: D = 0.5/M
#define M 10

// Structure representing an event
struct event {
double time;     // Event occurrence time
char type;       // Event type
char pclass;     // Class of the customer
involved in the event
double mem_size;     // Memory size
int num_iteration;     // Number of iteration
double time_req;     //Request occurence time
double time_start_mig; //Start migration time
double downtime;     //downtime
struct event *next;     // Pointer to the next
event in the list
};

// Structure representing a customer waiting
in the queue
struct customer {
double t_arr;    // Time of arrival
double mem_size;  // Memory size
struct customer *next;  // Pointer to the
next customer in the list
};

// Event list
struct event *event_list;
```

```
// List of waiting customers
struct customer *q[MAX_CLASSES];

// Global variables
int seed; // Random seed
int N;
int nt;
int C;
int k;
int tot_arrivals, tot_departures;
int arrivals[MAX_CLASSES], n_iter[M];
double now;   // Current time
double tot_w, tot_mig, tot_downtime, tot_iter;
double w[MAX_CLASSES], Tdw[MAX_CLASSES],
double Tmig[MAX_CLASSES], iter_num[MAX_CLASSES];
double lambda[MAX_CLASSES], mu[MAX_CLASSES];
double Res[M], Vmem[M];
double Tstar, Tx, Tdownstart, Sx, Star, Sres[M],
double Sr, Siter[M], Tdownstart_iter;
double Tres[M], Tr, load[M], Titer[M];
double V[M];
//double V[]={5,6};

// This function inserts a new event in the event
list ordered by time
// Arguments:
//  time = occurrence time of the new event
//  type = type of the new event
//  pclass = class of the customer
involved in the new event
//  service = service time of the arriving
customer, in case the event is
an arrival, 0 otherwise
void insert_new_event(double time,
char type, char pclass,
double mem_size, int num_iteration,
double time_req, double time_start_mig,
```

```
double downtime) {

// Declare some useful pointers
struct event *w1,*w2,*w3;

// Create the new event using the
 provided arguments
w3 = (struct event *) malloc(sizeof(struct event));
w3->time = time;
w3->type = type;
w3->pclass = pclass;
w3->mem_size = mem_size;
w3->num_iteration = num_iteration;
w3->time_req= time_req;
w3->time_start_mig = time_start_mig;
w3->downtime = downtime;

if (DEBUG) fprintf(stderr,"DEBUG 12: New
event object created: %d\n",w3);

// Insert the new event in the event list keeping the
correct order
if(event_list == NULL) {
w3->next = NULL;
event_list = w3;
if (DEBUG) fprintf(stderr,"DEBUG 13: Event
list was empty, new event added
 to list head: %d\n",event_list);
}
else if (event_list->time > w3->time) {
w3->next = event_list;
event_list = w3;
if (DEBUG) fprintf(stderr,"DEBUG 14:
Event list was not empty,
new event added to list head:
%d\n",event_list);
}
```

```
else {
w1 = event_list;
w2 = event_list->next;
while ((w2 != NULL) && (w2->time <= w3->time)) {
w1 = w2;
w2 = w2->next;
}
w1->next = w3;
w3->next = w2;
if (DEBUG) fprintf(stderr,"DEBUG 15:
Event list was not empty,
new event added to list\n",event_list);
}

if (DEBUG) fprintf(stderr,"DEBUG 01: New event inserted:
time = %1.6f, type = %s, class = %d,
mem_size = %1.6f, num_iteration = %d,
time_req = %1.6f, time_start_mig = %1.6f,
downtime = %1.6f, next = %d\n", w3->time,
(w3->type == 1 ? "MIGRATION_REQ" :
(w3->type == 2 ? "ITERATION" :
"END_MIGRATION")), w3->pclass,
 w3->mem_size, w3->num_iteration,
 w3->time_req, w3->time_start_mig,
 w3->downtime, w3->next);

}

// This function extracts the first event
from the event list
struct event * get_event() {

// Declare some useful pointers
struct event *w3;

// The list is empty and it should not be:
generate an error
```

```
if (event_list == NULL) {
fprintf(stderr,"ERROR: event list is
empty when it should not be\n");
exit(-1);
}

// Return the first event from the list and
update the list pointer
w3 = event_list;
event_list = w3->next;

if (DEBUG) fprintf(stderr,"DEBUG 02: Next event
extracted: time = %1.6f, type = %s,
class = %d, mem_size = %1.6f,
num_iteration = %d, time_req = %1.6f,
time_start_mig = %1.6f, downtime = %1.6f,
next = %d\n", w3->time, (w3->type == 1 ?
 "MIGRATION_REQ" : (w3->type == 2 ?
 "ITERATION" : "END_MIGRATION")),
 w3->pclass, w3->mem_size, w3->num_iteration,
 w3->time_req, w3->time_start_mig,
 w3->downtime, w3->next);

return w3;

}

// This function appends a new customer to the list of
waiting customers according to the class
// Arguments:
//  pclass = class of the customer
//  time = arrival time of the customer
//  service = service time of the customer
void append(char pclass, double time, double mem_size) {

// Declare some useful pointers
struct customer *w1,*w2,*w3;
```

```
// Create the customer using the provided arguments
w3 = (struct customer *) malloc(sizeof(struct customer));
w3->t_arr = time;
w3->mem_size = mem_size;

// Append the customer to the relevant
class waiting list
if(q[pclass] == NULL) {
w3->next = NULL;
q[pclass] = w3;
}
else {  // Append it to the end of the list
w1 = q[pclass];
w2 = q[pclass]->next;
while (w2 != NULL) {
w1 = w2;
w2 = w2->next;
}
w1->next = w3;
w3->next = w2;
}

if (DEBUG) fprintf(stderr,"DEBUG 03: New customer
added to queue %d: time = %1.6f,
service = %1.6f, next = %d\n",
pclass,w3->t_arr,w3->mem_size,w3->next);

}

// This function extracts the first customer
from the customer list
// Arguments:
//  pclass = class of the customer
struct customer * get_customer(char pclass) {
// Declare some useful pointers
struct customer *w3;
```

```
// The list is empty and it should not be: generate an error
if (q[pclass] == NULL) {
fprintf(stderr,"ERROR: customer list of class
%d is empty when it should not be\n",pclass);
exit(-1);
}

// Return the first customer from the list and update
the list pointer
w3 = q[pclass];
q[pclass] = w3->next;

if (DEBUG) fprintf(stderr,"DEBUG 04: Next customer
extracted from queue %d: time = %1.6f,
service = %1.6f, next = %d\n",pclass,
w3->t_arr,w3->mem_size,w3->next);

return w3;

}

// This function generates an instance of
an exponential random variable
// Arguments:
//  param = parameter of the exponential
distribution, i.e., inverse of the mean value
double expon(double param) {

// Declare some useful variables
int rnd_num;
double unif,val;

// Generate a uniform random number
 between 0 and 1
rnd_num = rand();
if (rnd_num == RAND_MAX) rnd_num--;
```

```
unif = (double)rnd_num/RAND_MAX;

// Transform the uniform random variable into an
exponential one using the inverse
unction rule
val = -log(1.0-unif)/param;
if (DEBUG) fprintf(stderr,"DEBUG 05: New exponential
random variable with parameter
%1.6f: generated value = %1.6f\n",param,val);
return val;

}

// This function generates an instance of the
service time random variable
// Arguments:
//  pclass = class of the customer
double serv(char pclass) {

// Returns an exponential service time
return expon(mu[pclass]);

}

//funzione di comparazione per qsort()
int comp (const void * a, const void * b){
    return ( *(double*)a - *(double*)b );
}
/*double min(double x, double y){
 if(x<=y) return x;
 else return y;
 }*/

double minima (double x[], int n){
    double minimo = x[0];
    for(int i=1; i<n; i++) {
        if(x[i] < minimo) minimo = x[i];
```

```
    }
    return minimo;
}

// Initialization function
void initialize() {

int j;

// Initialize all the global variables and counters
k = 0; tfree = 0.0; now = 0.0;
tot_arrivals = 0; tot_departures = 0; tot_w = 0.0;
 tot_mig = 0.0; tot_downtime = 0.0;
    for (int i=0; i<M; i++) {
        V[i]=rand()%100 + 1; //genero un numero
         casuale compreso tra 1 e 101
    }
for (j = 0; j < C; j++) {
arrivals[j] = 0; w[j] = 0.0; Tdw[j] = 0.0;
Tmig[j] = 0.0;
// Initialize the list of events by inserting
the first arrival for each class
        insert_new_event(expon(lambda[j]),
         MIGRATION_REQ,j,V[j], 0,0,0,0);
q[j]=NULL;
}
}

// Main program
int main(int argc, char* argv[]) {

   // Define some useful variables and pointers
    int j=0;
    int start_index = 0, inizio_indice=0;
    int l=0;
    nt = M;
    double t=0, x=0;
```

```
    Tx=0;
    Tdownstart=0;
    int r =M;
double Vth = 0.1;
struct event *e, *aux;
struct customer *c, *caux;
    int b=nt;
    double s=0;


// Check the command line arguments: must be
at least 6, including the simulator executable
// Otherwise generate an error
if (argc < 6) {
fprintf(stderr,"Usage: %s <seed> <N_samples>
<N_classes> <rho_class0>
<serv_class0> <rho_class1>
<serv_class1> ...\n",argv[0]);
exit(-1);
}

// Get the random generator seed from
the first argument
seed = atoi(argv[1]);
// If the provided seed is zero, use the
current timestamp
if (seed == 0) seed = time(NULL);
srand(seed);

// Get the number of customers to be
simulated from the second argument
N = atoi(argv[2]);

// Get the number of classes from the
third argument (no more than MAX_CLASSES)
C = atoi(argv[3]);
if (C > MAX_CLASSES) C = MAX_CLASSES;
```

```
// Check if there are enough arguments for
each class (load + average service
time for each class)
// Otherwise generate an error
if (argc < 2*C+4) {
fprintf(stderr,"Missing load and service time
 for classes %d to %d\n",(argc-4)/2,C-1);
exit(-1);
}
// Get the load and average service time for
each class from the following arguments
// and set mu = 1/service and lambda = rho*mu
for (j = 0; j < C; j++) {
mu[j] = 1.0/atof(argv[4+2*j+1]);
lambda[j] = atof(argv[4+2*j])*mu[j];
}
    //printf("\n# lambda \%f \n",lambda[j-1]);
// Initialize the simulator
initialize();

    //qsort(V, M, sizeof(double), comp);
    for (j=0; j<M; j++) {
        printf("\n # V[j] \%f \n", V[j]);
    }

// Start the main loop
while (tot_departures < N) {
if (DEBUG) {
fprintf(stderr,"DEBUG 17: Cycling...\n");
fprintf(stderr,"          Current event list:\n");
fprintf(stderr,"          ");
aux = event_list;
while (aux != NULL) {
fprintf(stderr,"[ addr = %d, time = %1.6f,
next = %d ]----->",
aux,aux->time,aux->next);
```

```
aux = aux->next;
}
fprintf(stderr,"\n");
}
if (DEBUG) {
fprintf(stderr,"          Current customer list:\n");
for (j=0;j<C;j++) {
fprintf(stderr,"          q[%d]----->",j);
caux = q[j];
while (caux != NULL) {
fprintf(stderr,"[ addr = %d,
time = %1.6f, next = %d ]----->",
caux,caux->t_arr,caux->next);
caux = caux->next;
}
fprintf(stderr,"\n");
}
}

// Start processing the first event in the list
and set the current time
e = get_event();
now = e->time;

if (DEBUG) fprintf(stderr,"DEBUG 10:
Current time = %1.6f\n",now);

if (e->type == MIGRATION_REQ && k==0) {
            k=k+M; tot_arrivals=tot_arrivals+M;
            arrivals[e->pclass]=arrivals[e->pclass]+M;
            qsort(V, M, sizeof(double), comp);
            j=0;
   if (DEBUG) fprintf(stderr,"DEBUG 06:
Arrival: time = %1.6f, class = %d,
mem_size = %1.6f\n",e->time,
e->pclass,e->mem_size) ;
            while (t==0) {
```

```
                for (j=0; j<M; j++) {
                    Res[j]=V[j];
                    Tres[j]=Res[j]* M;
                }
                Tx = minima(Tres,b);
                t=t+Tx;
            }
            while(nt!=0){
                for (int i=start_index; i<M; i++) {
                    Tr=Tx/nt;
                    Res[i]=Res[i]-Tr;
                }
                for (j=start_index; j<M; j++) {
                    if(Res[j]==0){
                        x=x+1;
                        n_iter[j]++;
                        Tstar=t-Titer[j];
                        Vmem[j] = Tstar*D;
                        Res[j]= Vmem[j];
                        Titer[j] = t;
                        if(Vmem[j]<Vth ){
                            nt--;
                            if(j==0){
                                Tdownstart=t;
                            }
                            else if(j==M-1 && tot_arrivals == N){
                                Tx=Res[j];
                                insert_new_event(now, END_MIGRATION,
                                 e->pclass, Res[j],x+1,
e->time_req,
e->time_start_mig,
Tdownstart);
                            }
                            else if(j==M-1){
                                Tx=Res[j];
                                insert_new_event(now +
                                 expon(lambda[e->pclass]),
```

```
ITERATION, e->pclass,
V[j],x+1, e->time_req,
e->time_start_mig,
Tdownstart);
                            }
                            else {
                                    Tx=Res[j];
                            }
                            start_index++;
                            Tres[j]=M*N*100000;
                        }
                    }
                }
                for (int i=start_index; i<M; i++) {
                    Tres[i]=Res[i]* nt;
                    Tx = minima(Tres, M);
                }
                t=t+Tx;
            }
            w[e->pclass] = w[e->pclass] +
             (t - e->time_req);
            if (k==M && tot_arrivals != N) {
                for (j=0; j<M; j++) {
                    insert_new_event(now +
                     expon(lambda[e->pclass]),
MIGRATION_REQ,
e->pclass, V[j], x+1,
t, t, Tdownstart);
                }
            }
            else {
                append(e->pclass, e->time, e->mem_size);
            }
        }
        else if (e->type == ITERATION) {
            tot_arrivals=tot_arrivals+M;
            tot_departures=tot_departures+M;
```

```
            arrivals[e->pclass]=arrivals[e->pclass]+M;
            j=0;
            while (s==0) {
                r=M;
                inizio_indice=0;
                for (j=0; j<M; j++) {
                    Res[j]=V[j];
                    Sres[j]=Res[j]* M;
                }
                Sx = minima(Sres,b);
                s=s+Sx;
            }
            while(r!=0){
                for (int i=inizio_indice; i<M; i++) {
                    Sr=Sx/r;
                    Res[i]=Res[i]-Sr;
                }
                for (j=inizio_indice; j<M; j++) {
                    if(Res[j]==0){
                        x=x+1;
                        n_iter[j]++;
                        Vmem[j] = Star*D;
                        Star=s-Siter[j];
                        Siter[j]=s;
                        Res[j]= Vmem[j];
                        if(Vmem[j]<Vth ){
                            r--;
                            if(j==M-1 && tot_arrivals == N){
                                Sx=Res[j];
                                insert_new_event(now,
                                 END_MIGRATION,
e->pclass, Res[j],x+1,
e->time_req,
e->time_start_mig,
Tdownstart);
                            }
                            else if(j==M-1){
```

```
                                    Sx=Res[j];
                                    s=-Sx;
                                    insert_new_event(now +
                                     expon(lambda[e->pclass]),
ITERATION, e->pclass,
V[j],x+1, e->time_req,
e->time_start_mig,
Tdownstart);
                                }
                                else {
                                    Sx=Res[j];
                                }
                                inizio_indice++;
                                Sres[j]=M*N*100000;
                            }
                        }
                    }
                    for (int i=inizio_indice; i<M; i++) {
                        Sres[i]=Res[i]* r;
                        Sx = minima(Sres, M);
                    }
                    s=s+Sx;
                    l=l+s;
                }
                append(e->pclass, e->time, e->mem_size);
        }
else if (e->type == END_MIGRATION) {
                k=k-M; tot_departures=tot_departures+M;
                x=x+1;
    if (DEBUG) fprintf(stderr,"DEBUG 16:
     Departure: time = %1.6f,
class = %d,
service = %1.6f\n",
e->time,e->pclass,
e->mem_size);
    Tmig[e->pclass] = Tmig[e->pclass] + (t - e->time_start_mig);
                Tdw[e->pclass] = Tmig[e->pclass] - Tdownstart;
```

```
    iter_num[e->pclass] = iter_num[e->pclass] + x ;
        }
if (DEBUG) fprintf(stderr,"DEBUG 08:
Removing event object\n");
if (e == NULL) {
fprintf(stderr,"ERROR:
event object is NULL
when it should not be\n");
exit(-1);
}
free(e);

if (DEBUG) fprintf(stderr,"DEBUG 09:
Current state = %d\n",k);
if (DEBUG) fprintf(stderr,"DEBUG 11:
Total arrivals = %d,
Total departures = %d\n",
tot_arrivals,tot_departures);
if (DEBUG) fprintf(stderr,"--------------
------------------------------
------------------------------
----------\n\n");
    } // End of the main loop

// Update and print some variables
for(j = 0; j < C; j++) {
        tot_mig = tot_mig + Tmig[j];
tot_downtime = tot_downtime + Tdw[j];
tot_iter = tot_iter + iter_num[j];
Tmig[j] = Tmig[j]/M;
        w[j] = Tmig[j]*((N/M)-1);
tot_w = tot_w + w[j];
Tdw[j] = Tdw[j]/M;
iter_num[j] = iter_num[j] /N;
        w[j] =w[j]/(N/M);
        load[j] = Tmig[j]* lambda[j];
}
```

```
tot_w = tot_w/(N/M);
tot_downtime = tot_downtime/M;
tot_mig = tot_mig / (M);
tot_iter = tot_iter / N;

for(j = 0; j < C; j++) {
        //printf("\n# load \%f \n",load[j]);
printf("\n# Class\t\tMean waiting time
\tMean migration time
\tMean downtime
\tMean number of iterations\n");
printf("  %d\t\t%1.6f\t\t%1.6f\t
\t%1.6f\t\t%1.6f\n",j,
w[j],Tmig[j],Tdw[j],iter_num[j]);
}
    printf("\n# x \%f \n",x);
printf("\n# Overall:\tMean waiting time
\tMean migration time
\tMean downtime
\tMean number of iterations\n");
printf("\t\t%1.6f\t\t%1.6f\t\t%1.6f\t\t%1.6f\n",
tot_w,tot_mig,tot_downtime,tot_iter);
}
```

Bibliografia

[1] R. Buyya Chee Shin Yeo S. Venugopal. Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities. *High Performance Computing and Communications.*, pages 5–13, 25-27 Sept. 2008.

[2] Timothy Grance Peter Mell. The nist definition of cloud computing. *NIST Special Publication 800-145 (Draft)*, January 2011.

[3] R. Uhlig G. Neiger D. Rodgers et al. Intel virtualization technology. *IEEE Computer Society*, May 2005.

[4] Techtarget.com. http://searchservervirtualization.techtarget.com/definition/virtual-machine, June 2013.

[5] A. Agarwal S. Raina. Live migration of virtual machines in cloud. *International Journal of Scientific and Research Publications*, II, June 2012.

[6] Tal Garfinkel Rosenblum, Mendel. Virtual machine monitors: Current technology and future trends. *Computer 38.5*, pages 39–47, 2005.

[7] https://en.wikipedia.org/wiki/hypervisor, June 2013.

[8] Al-Fares Mohammad A. Loukissas A. Vahdat. A scalable, commodity data center network architecture. *ACM SIGCOMM Computer Communication Review.*, 38(4), 2008.

[9] A. Greenberg S. Kandula D.A. Maltz J.R. Hamilton C. Kim P.Patel N. Jain P. Lahiri S. Sengupta. Vl2: A scalable and flexible data center network. *SIGCOMM '09 Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[10] Nelson Michael Beng-Hong Lim and Greg Hutchins. Fast transparent migration for virtual machines. *USENIX Annual Technical Conference, General Track*, 2005.

[11] Sapuntzakis Constantine P. et al. Optimizing the migration of virtual computers. *ACM SIGOPS Operating Systems Review*, pages 377–390, 2002.

[12] Clark Christopher et al. Performance and energy modeling for live migration of virtual machines. *Proceedings of the 2nd conference on Symposium on Networked Systems Design and Implementation*, II, 2005.

[13] Liu Haikun et al. Performance and energy modeling for live migration of virtual machines. *Proceedings of the 20th international symposium on High performance distributed computing. ACM.*, 2011.

[14] http://searchcio-midmarket.techtarget.com/definition/benchmark, June 2013.

[15] P. Svard B. Hudzia J. Tordsson E. Elmroth. Evaluation of delta compression techniques for efficient live migration of large virtual machines. 2011.

[16] S. Akoush R. Sohan A. Rice A. W. Moore A. Hopper. Predicting the performance of virtual machine migration. *Annual IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, 2010.

[17] F. Checconi T. Cucinotta M. Stein. Real-time issues in live migration of virtual machines. *Euro-Par 2009-Parallel Processing Workshops*, pages p. 454–466, 2010.

[18] Y. Du H. Yu G. Shi J. Chen W. Zhen. Microwiper: Efficient memory propagation in live migraztion of virtual machines. *39th International Conference on Parallel Processing*, 2010.

[19] Hines Michael R. Umesh Deshpande Kartik Gopalan. Post-copy live migration of virtual machines. *ACM SIGOPS operating systems review 43.3*, pages 14–26, 2009.

[20] Waldspurger Carl A. Memory resource management in vmware esx server. *ACM SIGOPS Operating Systems Review 36.SI*, pages 181–194, 2002.

[21] S. Norden G. Mainmaran C. Siva Ram Murthy. Dynamic planning based protocols for real-time communication in lan and switched lan environments.

[22] http://searchstorage.techtarget.com/definition/storage, June 2013.

[23] Ramakrishnan P. Shenoy J. Van der Merwe. Live data center migration across wans: a robust cooperative context aware approach. *Proceedings of the 2007 SIGCOMM workshop on Internet network management. ACM.*, 2007.

[24] Network Working Group. W. Simpson. Request for comments: 1853.

[25] et al. Travostino, Franco. Seamless live migration of virtual machines over the man/wan. *Future Generation Computer Systems 22.8*, pages 901–907, 2006.

[26] et al. Watanabe, Hidenobu. A performance improvement method for the global live migration of virtual machine with ip mobility. *Proc. the Fifth International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2010).*, 2010.

[27] http://en.wikipedia.org/wiki/osimodel, June 2013.

[28] http://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf, April 2013.

[29] A. Manzalini R. Minerva F. Callegati W. Cerroni A. Campi. Clouds of virtual machines in the edge networks.

[30] http://searchdatacenter.techtarget.com/definition/workload, June 2013.

[31] W. Cerroni F. Callegati. Live migration of virtualized edge networks: Analytical modeling and performance evaluation. 2013.

[32] P.Barham B.Dragovic K.Fraser S.Hand T.Harris A.Ho R.Neugebauer I.Pratt A.Warfield. Xen and the art of virtualization. *SOSP'03*, October 2003.

[33] D.Darsena G.Gelli A. Manzalini F. Melito. Live migration of virtual machines among edge networks via wan links. *Future Network Summit*, 2013.

[34] W.A. Gardner. *Introduction to Random Processes*. McGraw-Hill, 2nd edition edition, 1990.